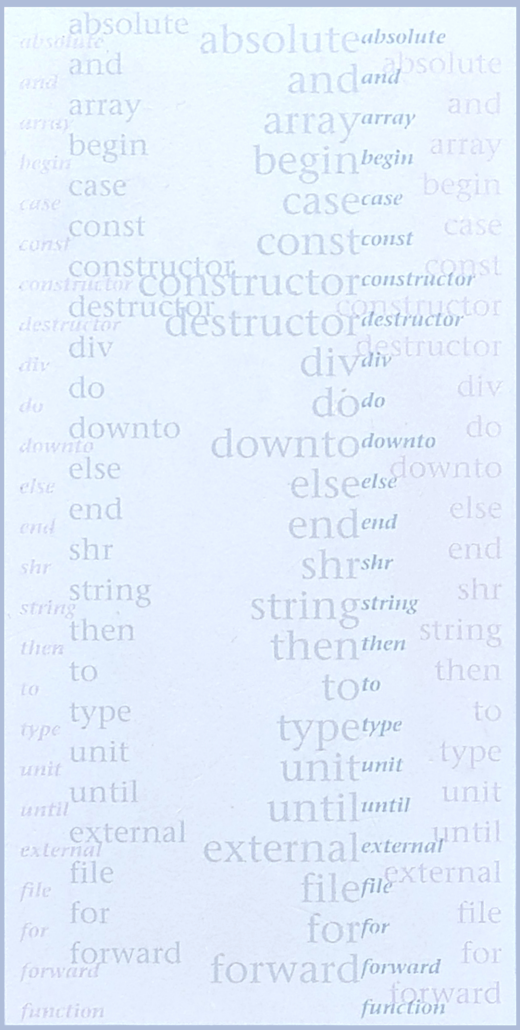
Turbo Pascal 6.0



Benutzerhandbuch

Das produktive Pascal-Entwicklungssystem für objektorientierte Programmierung. Komplett mit mausgesteuerter Benutzeroberfläche, integriertem Debugger und Assembler.

B O R L A N D

Turbo Pascal 6.0

Turbo Pascal 6.0

Benutzerhandbuch

B O R L A N D

Copyright © 1990 by Borland International, INC.
Jede Vervielfältigung dieses Handbuches sowie des Turbo Pascal Softwareprogrammes wird strafrechtlich verfolgt. Die Rechte an der Dokumentation und die Rechte an dem Softwareprogramm Turbo Pascal liegen bei Borland International.

Der rechtmäßige Erwerb der Programmdisketten und der Handbücher erlaubt die Nutzung der Programme analog der Benutzung eines Buches. Entsprechend der Unmöglichkeit, daß ein Buch zugleich an verschiedenen Orten von mehreren Personen gelesen wird, darf das Softwareprogramm Turbo Pascal nicht gleichzeitig von verschiedenen Personen an verschiedenen Orten und auf verschiedenen Geräten benutzt werden. Diskettenkopien dürfen lediglich zum Zweck der Datensicherung angefertigt werden.

Sie (der Lizenznehmer von Turbo Pascal) dürfen eigene Programme, die Sie mit dem Turbo-Pascal-Sprachcompiler geschrieben und compiliert haben, ohne Einschränkung oder Verpfichtung zu zusätzlichen Lizenzgebühren verwenden, weitergeben oder verkaufen, solange diese Programme einen Copyright-Vermerk enthalten. Unter Copyright-Vermerk verstehen wir entweder Ihren eigenen Copyright-Vermerk oder gegebenenfalls einen Hinweis wie »Created using © Borland International 1990«. Die Originaldisketten von Turbo Pascal beinhalten einige Dateien mit Hardware- und Zeichensatzinformationen, die von dem Standard-Grafik-Unit (GRAPH.TPU) verwendet werden. Diese Dateien mit den Suffixen .CHR und BGI unterliegen besonderen Nutzungsbeschränkungen. Sie dürfen nicht als Dateien weiterverkauft werden. Sie dürfen diese Dateien aber in Ihren eigenen mit Turbo Pascal geschriebenen und compilierten Programmen verwenden. Sie dürfen diese compilierten Programme auch weiterverkaufen, solange sie einen Copyright-Vermerk beinhalten.

Die Beispielprogramme von Turbo Pascal, die auf den Originaldisketten mitgeliefert werden, demonstrieren die verschiedenen Eigenschaften von Turbo Pascal. Sie dienen ausschließlich zu Lehrzwecken. Sie dürfen diese Beispielprogramme editieren oder für den eigenen Gebrauch modifizieren. Sie dürfen diese Programme jedoch nicht als Ganzes oder in Teilen in compilierter Form oder als Quelltext weitergeben oder verkaufen. Sie dürfen die Routinen aus den Beispielprogrammen in eigenen Anwendungsprogrammen verwenden, solange die resultierenden Programme nicht einem Bespielprogramm oder einem Teil eines Beispielprogrammes in Erscheinung oder Funktion gleichen.

Einschränkung der Gewährleistung
Es wird keine Garantie für die Richtigkeit des Inhaltes dieses Handbuches übernommen. Da sich Fehler, trotz aller Bemühungen, nie vollständig vermeiden lassen, sind wir für Hinweise jederzeit dankbar.

Herausgeber
Borland GmbH
Lindwurmstraße 88
Postfach 15 03 40
8000 München 2
Telefon 0 89 / 72 01 00
Telefax 0 89 / 77 93 39
Telex 5 21 26 37 mcm d

Umschlag und typografische Konzeption: Rudolf Paulus Gorbach

Grafiken: Sid Gastl, Andreas Höhne

Übertragung ins Deutsche: Dieter Strehle, Kapitel 1: Alois Stockklauser

PostScript-Belichtung: CD Computer & Dokumentations GmbH, Aalen

Druck und Bindung: Schoder Druck GmbH, Gersthofen

1. Auflage, Oktober 1990

Inhalt

Einleitung 1

Über die Dokumentation 2
Installation von Turbo Pascal 2
Die Datei README 4
Typographie 4
Fragen und Probleme – die Beratung der Borland GmbH 5

1. Die neue integrierte Entwicklungsumgebung 7

Die Komponenten 8
Das Menüzeile und die einzelnen Menüs 8
Fenster in Turbo Pascal 11
Dialog-Fenster 15
Der Start der integrierten Entwicklungsumgebung 17
Zur Funktion eines Compilers 19
Wo wird ein übersetztes Programm gespeichert? 20
Das zweite Programm 21
Das dritte Programm 23
Grafische Spielereien 24

2. Die sieben Elemente der Programmierung 27

Die Elemente der Programmierung 27
Ausgabe 28
Datentypen 30
Operatoren 37
Eingabe 41
Bedingte Ausführung 42
Schleifen 45
Prozeduren und Funktionen 49
Das achte Element: Fehlersichere Programme 55

3. Units, Overlays und andere Geheimnisse 59

Was bitte ist eine Unit? 59
Zur Struktur 60
Wie benutzt man eine Unit? 63
Die Standard-Units 65
Erweiterungen von TURBO.TPL: Das Programm TPUMOVER 66
Selbstgeschriebene Units 66
Units und mehrfach deklarierte Bezeichner 68
uses-Anweisungen im Implementations-Teil und gegenseitige Abhängigkeiten 70
Units und große Programme 73
Overlays 74
Die technischen Hintergründe 75
... und die praktischen Voraussetzungen 78
Die Unit Overlay im Überblick 80
Overlays in ».EXE«-Dateien 85
Zur Programmierung 86
Möglichkeiten und Grenzen 92

4. Programmierung für Fortgeschrittene 97

Datenstrukturen und Typen 97
Zeiger und dynamische Variablen 100
Dateioperationen 104
Textdateien 105
Typisierte Dateien 106
Untypisierte Dateien 107
PC-spezifische Programmierung 108
Aufrufe von MS-DOS 108
Direkte Steuerung des Bildschirms 110
Prozedur-Variablen anstelle von case 113

5. Die Verwaltung von Projekten 117

Zum Thema Programmorganisation 117
Initialisierung 120
Die Menüpunkte Build und Make 120
Die Funktion Make 121
Die Funktion Build 122
Das Programm MAKE.EXE 123
Ein Beispiel 124

6. Fehlersuche und der integrierte Debugger 127

Fehlerklassen 127
Compilierungsfehler 127
Laufzeitfehler 128

Logische Fehler 128
Klassische Methoden der Fehlersuche 129
Fehlerprüfungen der Ein-/Ausgabe 129
Bereichsüberschreitungen 130
Fehlerstellen einkreisen 131
Der integrierte Debugger 132
Teil I: Eine schrittweise Einführung 133
Einige Worte zum Funktionsprinzip 133
... und eine kurze Übersicht der Möglichkeiten 134
Ein Beispiel 134
Zurück zum Programm 139
Watch-Ausdrücke 141
Zurück zum Programm 143
Fehlerkorrekturen und Neustart 145
Ein letztes Mal: Zurück zum Programm 147
Fehlersuche in größeren Projekten 147
Teil II: Die Befehle des Debuggers 149
Tastenbefehle des Debuggers und verwandte Kommandos 149
Ausdrücke für Watch und Evaluate 151
Teil III: Grenzbedingungen und Programmentwicklung 157
Programmierung »mit dem Debugger« 158
Rekursive Routinen 159
Zum wirklich effektiven Prüfen von Programmen... 163
Ein standardisierter Ansatz 164
Modifikationen verlangen doppelte Aufmerksamkeit 165
Defensiver Programmierstil 165
Fehlersuche von »unten nach oben« 166
Suche nach gleichen und ähnlichen Fehlern 166

7. Die IDE-Referenz 167

Aufrufen und Verlassen von Turbo Pascal 168
Kommandozeilen-Optionen 168
Verlassen vonTurbo Pascal 171
System-Menü 171
File-Menü 171
Edit-Menü 175
Search-Menü 176
Run-Menü 180
Compile-Menu 182
Debug-Menü 184
Options-Menü 188
Window-Menü 196
Help-Menü 198

8. Der Editor von A bis Z 201

Der alte und der neue Editor 201
Editor-Referenz 202
Cursorsprünge 204
Blockkommandos 204
Weitere Editor-Kommandos 206
Suchen und Ersetzen 207
Begrenzerpaare suchen 208

9. Die Kommandozeilen-Version des Compilers 211

Hilfe? Hilfe! 211
Der Aufruf von TPC.EXE... 212
Parameter und Schalter 213
Compiler-Befehle 214
Der bedingte Compiler-Befehl DEFINE 215
Die Art der Compilierung 216
Verzeichnisse und Suchwege 218
/V – Der Schalter für den Turbo Debugger 219
Parameter für den Linker 220
Die Datei TPC.CFG 221
Compilieren im Protected Mode 222

10. Die Zusatzprogramme 223

BINOBJ – Konvertierung in .OBJ-Dateien 223
GREP – Absuchen von Quelltexten 225
Dateinamen 226
Die Schalter von GREP 226
Suchstrings und reguläre Ausdrücke 227
Einige Beispiele 229
MAKE.EXE – die automatische Verwaltung von Programmen 232
Die Erzeugung von MAKE-Dateien 232
Der Aufruf von MAKE 248
Fehlermeldungen von MAKE 250
Fehler mit sofortigem Abbruch »Fatal Errors« 250
Fehler »Errors« 251
THELP – Syntax-Hilfe für alle ASCII-Editoren 253
Der Start von THELP 253
... und die Aktivierung 254
Entfernen von THELP 255
Einfügen von Texten der Hilfestellung ins laufende Programm 255
Kommandozeilen-Parameter 255
Details zu den Kommandozeilen-Parametern 256
TOUCH – »Erneuern« von Dateien 258
TPUMOVER – Die Bibliotheksverwaltung von Turbo Pascal 259

Einige technische Details 259
Die Verwendung von TPUMOVER 260

A. Unterschiede der Versionen 261

Compilierung von Programmen der Version 4.0 261
Allgemeines 261
.TPM und .MAP-Dateien 262
Nullstrings und die Funktion Length 262
Gleitkommazahlen 262
Units 263
Compiler-Befehle im Quelltext 263
Lokale und globale Variablen 264
Der Aufruf von TPC.EXE 264
Compilierung der Toolboxen 265
Modifizierte Prozeduren und Funktionen 265
Erweiterungen der Laufzeitbibliothek 266
Syntax- und Laufzeitfehler 267
Konvertierung von Programmen der Version 3.0 268
Die Arbeit mit Turbo Pascal 269
Die Implementation des Compilers 270
Deklarationen 271
Compiler-Befehle 272
Eingaben über die Tastatur 273
Vordefinierte Bezeichner 275
Programmiertechniken 276
Erweiterungen und Verbesserungen 280
Die Units Turbo3 und Graph3 283
Turbo3 284
Graph3 285
Die Konvertierung im Detail 285
Vordefinierte Bezeichner 285
Datentypen 286
Sprachstruktur 288
Ein- und Ausgabe 288
Organisation des Programms und Speicherbelegung 289
Compiler-Befehle und Fehlerprüfung 290
Maschinensprache-Routinen 291

B. Unterschiede zu ANSI-Pascal 293

Abweichungen von den ANSI-Forderungen 293
Erweiterungen des ANSI-Standards 295
Abhängigkeiten von der Implementation 298
Behandlung von Fehlern 299

Index 301

Einleitung

Turbo Pascal erfüllt die Wünsche von Benutzern mit den unterschiedlichsten Anforderungen an einen Compiler. Turbo Pascal ist eine strukturierte Hochsprache, mit der Sie Programme für jeden Anwendungstyp und jede Anwendungsgröße schreiben können. Turbo Pascal 6.0 ist für den IBM PC geschrieben und auf den Modellen PC, XT, AT, PS/2 sowie allen wirklich »Kompatiblen« lauffähig. Turbo Pascal 6.0 basiert auf Borlands früheren Versionen, dem weltweiten Pascal-Standard. Die Version 6.0 enthält zum einen sämtliche Möglichkeiten früherer Versionen – zum anderen bietet sie:

- die neue Borland-Entwicklungsumgebung (IDE) mit
  - mehreren überlappenden Fenstern
  - Mausunterstützung
  - einem Multidatei-Editor, der in der Lage ist, Dateien mit bis zu 32767 Zeilen zu bearbeiten
  - einer Arbeitsfläche, die vollständig abgespeichert und wiederhergestellt werden kann
  - erweiterten Debugging-Möglichkeiten
- Turbo Vision: objektorientierte Tools (die Tools, mit denen wir die IDE geschrieben haben)
- einen integrierten Assembler
- private Felder und Methoden in Objekt-Deklarationen
- erweiterte Syntax-Direktiven (`$X`), damit Sie Funktionen wie Prozeduren behandeln können (und deren Ergebnis damit ignorieren dürfen)
- 286-Code-Generierung (`$G`)
- Adreßreferenzen in typisierten Konstanten
- Far- und Near-Prozedurdirektiven (`$F`)
- Integration initialisierter Daten (`$L`) aus Objekt-Dateien
- einen neuen Heap-Manager, der schneller arbeitet und die Fragmentierung des Heapspeichers reduziert (*FreeMin* und *FreeList* wurden ersetzt, in Kapitel 14 »Interne Details« des *Programmierhandbuches* finden Sie weitere Informationen)
- Hypertext-Hilfe mit Programmbeispielen zu praktisch jeder Prozedur und Funktion

Über die Dokumentation

Die vier Handbücher der Turbo-Pascal-Dokumentation dienen verschiedenen Zwecken

- Das *Benutzerhandbuch* ist größtenteils im Stile eines Lehrbuches gehalten. Es erklärt die Arbeit mit der integrierten Entwicklungsumgebung und dem Kommandozeilen-Compiler, beschäftigt sich mit den Grundlagen der Programmiersprache Pascal und geht schließlich auf spezifische Dinge wie Management großer Projekte und die Fehlersuche mit einem Debugger ein.

- Das *Programmierhandbuch* beschreibt alle technischen Aspekte von Turbo Pascal. Hier finden Sie detaillierte Beschreibungen der Sprachspezifikation, der Standardbibliotheken sowie deren Implementierung. Dieses Buch enthält auch die Beschreibung des integrierten Assemblers.

- Im *Referenzhandbuch* finden Sie ein alphabetisches Verzeichnis aller Routinen, die Turbo Pascal zur Verfügung stellt. Es enthält außerdem eine Beschreibung der Compiler-Befehle und führt auch die Fehlermeldungen und -codes auf.

- Der Band *Objektorientierte Programmierung und Turbo Vision* gliedert sich in zwei Teile, von denen der erste Ihnen die objektorientierten Erweiterungen von Turbo Pascal vorstellt. Der zweite Teil behandelt Turbo Vision, einen völlig neuen Zugang zur Entwicklung von Anwendungsprogrammen.

Installation von Turbo Pascal

Turbo Pascal wird mit dem Installationsprogramm INSTALL ausgeliefert. Sie sollten INSTALL benutzen, damit alle Dateien, die Sie brauchen, an den richtigen Platz kopiert werden. Die Dateien von Turbo Pascal sind auf den Original-Disketten in archivierter Form (Datei-Extension .ZIP) abgespeichert und werden von INSTALL dearchiviert. Install speichert die dearchivierten Dateien in den notwendigen Verzeichnisse auf der Festplatte. Während der Installation wird eine entsprechende Konfiguration für die IDE wie auch für die Kommandozeilen-Version des Compilers erzeugt. Die Installation ist selbsterklärend. Wenn Sie schon einmal eine frühere Version von Turbo Pascal installiert haben, sind Sie mit dem Vorgang vertraut.

Wenn Sie nicht genau wissen, wie DOS-Befehle verwendet werden, sollten Sie zunächst in Ihren DOS-Handbüchern nachlesen, bevor Sie mit der Installation beginnen. Wir nehmen an, daß Ihnen DOS-Befehle vertraut sind. Sie sollten etwa DISKCOPY benutzen, um Arbeitskopien Ihrer Turbo-Pascal-Disketten anzufertigen. Bewahren Sie die Originaldisketten an einem sicheren Ort auf.

Lesen Sie die Borland Lizenzvereinbarung, die Sie zusammen mit Ihrem Turbo-Pascal-Paket bekommen haben.

Installation

1. Legen Sie die Installationsdiskette in Laufwerk A.

2. Geben Sie den Befehl `A:INSTALL`ein, und drücken Sie Enter.

3. Drücken Sie Enter, wenn Sie den Installationsbildschirm sehen, und folgen Sie dann den Anweisungen.

INSTALL arbeitet sowohl mit Festplattensystemen als auch mit Systemen, die mit zwei Diskettenlaufwerken ausgerüstet sind. Falls Sie sich dazu entschließen, die Turbo-Pascal-Dateien manuell zu dearchivieren, so verwenden Sie das Programm UNZIP, das Sie auf der Installationsdiskette finden. Um beispielsweise die Dateien TPC.EXE sowie TURBO.TPL aus der Datei TURBO.ZIP, die sich auf der Diskette in Laufwerk A befindet, zu dearchivieren, geben Sie folgenden Befehl ein:

```
unzip a:turbo turbo.tpl tpc.exe
```

Dadurch werden die Dateien TPC.EXE und TURBO.TPL extrahiert und im aktuellen Verzeichnis gespeichert, wobei allerdings vorausgesetzt wird, daß sich UNZIP.EXE bereits im aktuellen Verzeichnis befindet. Wenn die Installation erfolgreich abgeschlossen worden ist, erinnert Sie INSTALL nochmals daran, die Datei README zu lesen, da dort die aktuellsten Informationen aufgeführt sind. INSTALL informiert Sie auch darüber, welche Parameter die Dateien CONFIG.SYS und AUTOEXEC.BAT enthalten sollten, damit die Arbeit mit Turbo Pascal reibungslos verläuft.

Nach der Installation empfehlen wir, daß Sie sich zunächst einmal mit TPTOUR näher befassen. TPTOUR ist eine Führung durch die Highlights der neuen integrierten Entwicklungsumgebung (IDE). Sie finden es im Unterverzeichnis TOUR des Pascal-Verzeichnisses.

Nachdem Sie sich TPTOUR angesehen haben, werfen Sie einen Blick auf die neue Entwicklungsumgebung selbst. Wechseln Sie in das Verzeichnis, indem Sie Turbo Pascal installiert haben, tippen Sie `TURBO`, und drücken Sie Enter. Die Entwicklungsumgebung verlassen Sie mit Alt X. Sie können aber ebenso zunächst die restlichen Abschnitte dieser Einführung lesen.

Turbo Pascal konfigurieren

Die neue integrierte Entwicklungsumgebung erlaubt eine vollständige Konfiguration, ohne daß das Progrmm verlassen werden muß. Sie können Farben, Optionen, Präferenzen und auch einige Kommandozeilen-Optionen definieren.

Soll die IDE die Arbeitsfläche sichern, wenn Sie zwischen Ihren Sitzungen unterbrechen möchten, so setzen Sie im Fenster *Preferences* (*Options/Environment*) die Option *Autosave* auf ON (sowohl für *Environment* als auch für *Desktop*).

Laptop-Systeme

Bei Verwendung eines Laptop-Systems mit LCD- oder Plasma-Anzeige benötigen Sie entsprechende Bildschirmparameter. Geben Sie auf Betriebssystemebene `MODE BW80`ein, bevor Sie Turbo Pascal starten. Diese und weitere Optionen werden in Kapitel 7 »Die IDE-Referenz« erläutert.

Sie können natürlich eine Batch-Datei verwenden, um sich diese Arbeit zu

erleichtern. Noch einfacher ist es, Turbo Pascal selbst zu konfigurieren. Definieren Sie Ihre Einstellungen im Fenster *Startup options* (*Options/Environment/Startup*) oder starten Sie Turbo Pascal mit der Kommandozeilen-Option `/L`.

Die Datei README

Die Datei README enthält Informationen, die nicht mehr in die Handbücher aufgenommen werden konnten. Sie verzeichnet auch die Dateien auf den Disketten und enthält eine Kurzbeschreibung des Inhalts der Dateien. README lesen Sie wie folgt:

Wenn Sie Turbo Pascal noch nicht installiert haben:

1. Legen Sie die Installationsdiskette in Laufwerk A.

2. Tippen Sie `A:` ein, und drücken Sie Enter.

3. Tippen Sie dann `README`, und drücken Sie Enter. Mit den Pfeiltasten bewegen Sie sich durch die Datei, mit Esc verlassen Sie sie wieder.

Wenn Sie Turbo Pascal schon installiert haben:

1. Starten Sie Turbo Pascal aus dem Verzeichnis, in dem Sie es installiert haben, indem Sie `TURBO` eingeben und Enter drücken.

2. Drücken Sie F3, geben Sie `README` ein, und drücken Sie dann Enter. Turbo Pascal öffnet die Datei README in einem Editorfenster.

3. Wenn Sie README gelesen haben, schließen Sie das Editorfenster mit Alt F3. Sie können ebenso mit Alt X Turbo Pascal wieder verlassen.

Typographie

Zur Verdeutlichung verschiedener Zusammenhänge verwenden wir in diesem Handbuch mehrere unterschiedliche Schriftarten:

`Schreibmaschine` steht für Syntaxdefinitionen, Kommandozeilen-Parameter, Eingaben des Benutzers, Programmlistings und Ausgaben, die von Beispielprogrammen erzeugt werden;

`Schreibmaschine fett` kennzeichnet reservierte Wörter von Turbo Pascal in Listings wie `begin`, `end`, `program` usw.;

Fettschrift wird für reservierte Wörter (wie **begin**, **end** und **program**) im laufenden Text benutzt;

Kursiv für Namen von Variablen und anderen Bezeichnern im laufenden Text;

Esc für die Tasten Ihrer Tastatur, wie beispielsweise in »Ein Druck auf Esc entfernt das Menüfenster wieder vom Bildschirm«;

←, ↑ für die Pfeiltasten zur Bewegung des Cursors und andere Spezialtasten.

Fragen und Probleme – die Beratung der Borland GmbH

Bei Problemen mit *Turbo Pascal* bitten wir Sie, zuerst die Handbücher und die Datei README (auf der Turbo-Pascal-Diskette 1) sorgfältig durchzulesen. Wenn danach Fragen offenbleiben, können Sie sich an eine der folgenden Stellen wenden:

1. Ihren Softwarehändler oder einen Benutzerclub

2. In schriftlicher Form an:
Borland GmbH
Technische Beratung
Postfach 150340
Lindwurmstr. 88
8000 München 2

3. In sehr dringenden Fällen an die *Borland GmbH Hotline*:
Tel.: 089/7258001
Montag bis Donnerstag von 14 bis 17 Uhr
Freitag von 14 bis 16 Uhr

Sie sollten in allen Fällen die Handbücher sowie die folgenden Informationen parat haben:

1. Die Versionsnummer von Turbo Pascal. (Die Kommandozeilen-Version von Turbo Pascal gibt diese Nummer beim Start aus, innerhalb der integrierten Entwicklungsumgebung öffnet die Eingabe von Alt Leertaste das System-Menü: der darin verfügbare Befehl *About...* öffnet ein Fenster mit der entsprechenden Meldung.)

2. Produktnamen und Modellbezeichnung Ihres Computers sowie die Namen eventueller Zusatz-Hardware – insbesondere Grafik-Karten.

3. Den Namen des Betriebssystems (PC-DOS oder MS-DOS) sowie die entsprechende Versionsnummer. (Diese Nummer erhalten Sie durch die Eingabe des Befehls VER auf der DOS-Kommandoebene.)

1. Die neue integrierte Entwicklungsumgebung

Turbo Pascal ist weit mehr als nur ein schneller Pascal-Compiler – der Turbo-Pascal-Compiler ist in eine sehr komfortable Entwicklungsumgebung eingebunden, die mit einem Editor, einem Linker, einem Debugger, Funktionen zum Laden und Speichern von Texten und noch vielem mehr ausgestattet ist. Sämtliche Funktionen dieser integrierten Entwicklungsumgebung (im weiteren kurz IDE genannt) sind miteinander verbunden und von einem Punkt aus erreichbar: dem Hauptmenü.

Das Turbo Pascal-Programmierpaket enthält gleich zwei Compiler – einen in der integrierte Entwicklungsumgebung und eine Kommandozeilen-Version. Da der Umgang mit der integrierten Entwicklungsumgebung aber wesentlich einfacher ist, beschäftigen wir uns auch hauptsächlich mit dieser Variante.

Einem sofortigen Einstieg steht somit nichts im Wege, und Sie können damit beginnen, Ihr erstes Pascal-Programm mit Hilfe des in die IDE eingebauten Compilers zu entwickeln. Am Ende dieses Kapitels werden Sie

- gelernt haben, sich in der Entwicklungsumgebung zurechtzufinden;
- drei kurze Programme geschrieben und gespeichert haben;
- einige grundlegende Programmierkenntnisse erworben haben.

Die *Hilfestellung* von Turbo Pascal ist kontextbezogen und steht Ihnen in fast jeder Situation nach nur einem Tastendruck (F1 oder einem Maus-Klick) zur Verfügung, es sei denn, Ihr eigenes Programm hat die Steuerung übernommen. Das Hilfe-Menü (Alt H) bietet Ihnen das Inhaltsverzeichnis des Hilfe-Systems, einen ausführlichen Index, Suchfunktionen (Ctrl F1), die Fähigkeit zuvor gewählte Hilfetexte erneut aufzurufen (Alt F1) und, wenn das Hilfesystem bereits aktiviert ist, *Hilfe über Hilfe* (F1). Bildschirme der Hilfestellung können einen oder mehrere hervorgehoben dargestellte Punkte enthalten, die auf weitere Informationen verweisen.

Detailliertere Informationen über die IDE finden Sie in Kapitel 7 »Die IDE-Referenz«

Die Komponenten

In der integrierten Entwicklungsumgebung finden Sie drei sichtbare Komponenten: das Auswahlmenü in der obersten Zeile, die eigentliche Arbeitsfläche und darunter die Statuszeile. Viele Wahlpunkte aus dem Menü dienen zum Öffnen von Dialogfenstern. Bevor wir dazu übergehen, die einzelnen Wahlpunkte detailliert zu beschreiben, befassen wir uns mit den generischen Komponenten.

Hinweis: Menü-Befehle werden in den Handbüchern oft in abgekürzter Form verwendet: *D/W/Add Watch* steht zum Beispiel für *Debug/Watch/Add Watch*.

Das Menüzeile und die einzelnen Menüs

Sämtliche Menü-Befehle sind über die Menüzeile erreichbar. Die Menüzeile ist immer sichtbar, es sei denn, auf dem Bildschirm werden gerade die Ausgaben Ihres Programms und nicht die IDE dargestellt. Wenn das Auswahlmenü aktiviert ist, sehen Sie einen hervorgehoben dargestellten Menütitel; dieser zeigt das aktuell *gewählte* Menü an.

Wenn Sie einen Menü-Befehl mit Fortsetzungspunkten (...) wählen, öffnet sich ein Dialogfenster. Wählen Sie hingegen einen Befehl dem ein Pfeil folgt, gelangen Sie in ein weiteres Menü (ein Pop-Up-Menü). Menü-Befehle ohne Zusatzzeichen führen ihre Aufgaben sofort nach Anwahl aus. Im folgenden wird die Anwahl der Menü-Befehle über die Tastatur erläutert:

1. Drücken Sie F10. Dadurch wird die Menüzeile aktiviert.

2. Ein Menü wählen Sie mit den Pfeiltasten und Enter, oder durch die Eingabe des Tastenkürzels, d.h. den hervorgehobenen Buchstaben des gewünschten Menütitels. Mit Esc brechen Sie eine Option ab.

   Das Edit-Menü können Sie beispielsweise von der Menüzeile aus mit E, von jeder beliebigen anderen Stelle aus mit Alt E aufrufen.

3. Wählen Sie innerhalb des Menüs nochmals mit Hilfe der Pfeiltasten einen Befehl an und drücken Sie Enter bzw. geben Sie das Tastenkürzel ein (den hervorgehobenen Buchstaben des Befehls).

   Turbo Pascal führt dann entweder den Befehl aus, öffnet ein Dialogfenster oder öffnet ein weiteres Menü.

Sämtliche Befehle lassen sich auch mit der Maus anwählen:

1. Öffnen Sie das Menü, indem Sie den Menütitel anklicken.

2. Klicken Sie den Befehl an.

Sie können den Mauszeiger vom Menütitel natürlich auch geradlinig zum gewünschten Menü-Befehl ziehen und dann die Maustaste loslassen. (Dieser Vorgang wird abgebrochen, wenn Sie den Mauszeiger aus dem Menü ziehen und ihn dann loslassen.)

Die Menü-Befehle selbst sind nur dann verfügbar, wenn ihre Anwahl auch sinnvoll ist. Für den Aufruf der Hilfestellung ist es allerdings unerheblich, ob Sie einen nicht verfügbaren Befehl anwählen (hervorheben) – der Hilfetext wird auf jeden Fall ausgegeben.

Hinweis: In Turbo Pascal findet nur die linke Maustaste Verwendung. Wenn Sie lieber mit der rechten Taste arbeiten oder andere Maus-Einstellungen verändern möchten, so können Sie die Maus-Einstellungen in *Options/Environment/Mouse* Ihren Wünschen gemäß anpassen.

Tastenkürzel

Turbo Pascal bietet eine Reihe von Möglichkeiten zur schnellen Anwahl von Menü-Befehlen. Der schnellste Weg für Maus-Benutzer sieht folgendermaßen aus: halten Sie die Maustaste gedrückt, ziehen Sie den Mauszeiger vom Hauptmenü zum gewünschten Menü-Befehl und lassen Sie dann die Maustaste los; der Menü-Befehl wird augenblicklich ausgeführt. Noch schneller erfolgt die Anwahl eines Menü-Befehls, wenn Sie die in der Statuszeile dargestellten Tastenkürzel anklicken.

Tastatur-Benutzern stehen für denselben Zweck zahlreiche Tastenkürzel zur Verfügung. Sie erreichen bzw. aktivieren Menü-Befehle durch das gleichzeitige Drücken der Alt -Taste und des hervorgehobenen Buchstabens. Ist ein Menü aktiviert, dann brauchen Sie lediglich den hervorgehobenen Buchstaben des gewünschten Menü-Befehls (oder das nebenstehende Tastenkürzel) zu drücken. Innerhalb von Dialogfenstern arbeiten Sie auf die gleiche Weise.

In den nachstehenden Tabellen finden Sie die gebräuchlichsten Turbo-Pascal-Tastenkürzel:

Tabelle 1.1 Allgemeine Tastenkürzel

Taste	Menü-Befehl	Funktion
F1	*Help*	Ruft die Hilfestellung auf.
F2	*File/Save*	Speichert die Datei, die sich im aktiven Fenster befindet.
F3	*File/Open*	Öffnet das Dialogfenster *File* zum Laden einer Datei.
F4	*Run/Go to cursor*	Führt das Programm bis zu dem Punkt, an dem sich der Cursor befindet, aus.
F5	*Window/Zoom*	Vergrößert das aktive Fenster.
F6	*Window/Next*	Wechselt zum nächsten Fenster und aktiviert es.
F7	*Run/Trace into*	Startet das Programm im Debugger-Modus und führt die Unterprogramme im Einzelschritt-Modus aus.
F8	*Run/Step over*	Startet das Programm im Debugger-Modus ohne schrittweises Verfolgen der Unterprogramme.
F9	*Compile/Make*	Führt MAKE durch (compilieren und linken)
F10		Aktiviert die Menüzeile.

Tabelle 1.2 Menü-Tastenkürzel

Taste(n)	Menü-Befehl	Funktion
Alt Leerz.	System	Öffnet das System-Menü.
Alt C	Compile	Öffnet das Compile-Menü.
Alt D	Debug	Öffnet das Debug-Menü.
Alt E	Edit	Öffnet das Edit-Menü.
Alt F	File	Öffnet das File-Menü.
Alt H	Help	Öffnet das Help-Menü.
Alt O	Options	Öffnet das Options-Menü.
Alt R	Run	Öffnet das Run-Menü.
Alt S	Search	Öffnet das Search-Menü.
Alt W	Window	Öffnet das Window-Menü.
Alt X	File/Exit	beendet Turbo Pascal (zurück zu DOS).

Tabelle 1.3 Edit-Tastenkürzel

Taste(n)	Menü-Befehl	Funktion
Ctrl Del	*Edit/Clear*	Löscht den markierten Text aus dem Fenster, ohne ihn in die Zwischenablage zu kopieren.
Ctrl Ins	*Edit/Copy*	Kopiert den markierten Text in die Zwischenablage.
Shift Del	*Edit/Cut*	Kopiert den markierten Text in die Zwischenablage und löscht die Markierung.
Shift Ins	*Edit/Paste*	Kopiert den Text aus der Zwischenablage ins aktive Fenster.
Ctrl L	*Search/Search again*	Wiederholt den letzten Suchen- bzw. Ersetzen-Befehl.
Alt S R	*Search/Replace*	Öffnet das Dialogfenster für Suchen und Ersetzen.
Alt S F	*Search/Find*	Öffnet das Dialogfenster für Finden.
F2	*File/Save*	Sichert die Datei im aktiven Editorfenster.
F3	*File/Open*	Öffnet das Dialogfenster zum Öffnen einer Datei.

Tabelle 1.4 Tastenkürzel zur Fensterverwaltung

Taste(n)	Menü-Befehl	Funktion
Alt #		Zeigt ein Fenster, an, wobei # für die Nummer des gewünschten Fensters steht.
Alt 0	*Window/List*	Zeigt eine Liste aller geöffneten Fenster.
Alt F3	*Window/Close*	Schließt das aktive Fenster.
Alt F5	*Window/User screen*	Wechselt zum Benutzer-Bildschirm.
Shift F6	*Window/Previous*	Wechselt zum letzten Fenster und aktiviert es.
F5	*Window/Zoom*	Vergrößert/verkleinert das aktive Fenster.
F6	*Window/Next*	Wechselt zum nächsten Fenster u. aktiviert es.
Ctrl F5	*Window/Size*/Move	Ändert Größe und Position des aktiven Fensters.

Tabelle 1.5 Tastenkürzel für das integrierte Hilfe-System

Taste(n)	Menü-Befehl	Funktion
F1	*Help/Contents*	Öffnet das Hilfesystem kontextbezogen.
F1 F1	*Help/Help on help*	Öffnet das Menü Help-on-Help (Wenn das Hilfesystem bereits aktiviert ist, drücken Sie lediglich F1).
Shift F1	*Help/Index*	Öffnet den Hilfe-Index.
Alt F1	*Help/Previous topic*	Öffnet den zuletzt aktiven Hilfebildschirm.
Ctrl F1	*Help/Topic search*	aktiviert die Syntaxhilfe (nur im Editor).

Tabelle 1.6 Tastenkürzel für Debugger und Run

Taste(n)	Menü-Befehl	Funktion
Alt F9	*Compile/Compile*	Compiliert die aktuelle Datei
Ctrl F2	*Run/Program Reset*	Führt einen Programm-Reset durch.
Ctrl F4	*Debug/Evaluate/Modify*	Wertet einen Ausdruck aus.
Ctrl F7	*Debug/Add Watch*	Fügt einen Watch-Ausdruck hinzu.
Ctrl F8	*Debug/Toggle Breakpoint*	Setzt oder löscht einen bedingten Breakpoint.
Ctrl F9	*Run/Run*	Startet das Programm.
F4	*Run/Go To Cursor*	Führt das Programm bis zur Cursorposition aus.
F7	*Run/Trace Into*	Startet das Programm im Debugger-Modus und führt die Unterprogramme im Einzelschritt-Modus aus.
F8	*Run/Step Over*	Startet das Programm im Debugger-Modus ohne schrittweises Verfolgen der Unterprogramme.
F9	*Compile/Make*	Führt MAKE durch (compilieren und linken).

Fenster in Turbo Pascal

Die Arbeit in der IDE von Turbo Pascal erfolgt fast ausschließlich in *Fenstern*. Ein Fenster ist ein Teil des Bildschirms, den Sie verschieben, vergrößern und verkleinern, teilen, mit anderen Fenstern verdecken, öffnen und schließen können.

Die Anzahl der Fenster, die Sie öffnen können, hängt nur vom verfügbaren Speicher ab – *aktiv* ist allerdings immer nur ein einziges. Als aktives Fenster wird immer das Fenster bezeichnet, in dem Sie gerade arbeiten. Alle Kommandos und Texteingaben beziehen sich nur auf dieses Fenster. Öffnen Sie ein und dieselbe Datei jedoch in mehreren Fenstern, so wirken sich Aktionen innerhalb eines Fensters auch auf die anderen Fenster aus. Die Zahl der Fenster, die gleichzeitig geöffnet sein können, kann durch die Vergrößerung des Heap in der Startup-Option erhöht werden (*Options/Environment*).

Zwar gibt es mehrere Fenstervarianten, die meisten davon haben aber folgende Elemente gemeinsam:

- einen Titel
- ein Schließfeld
- einen Rollbalken
- ein Eckfeld zur Änderung der Größe
- ein Zoom-Feld
- eine Fensternummer

Das aktive Fenster erkennen Sie an der doppelten Umrahmung. Es verfügt jeweils über ein Schließfeld, ein Zoom-Feld, einen Rollbalken und ein Eckfeld über das die Fenstergröße geändert werden kann. Überlappen sich zwei Fenster, ist immer das Fenster im Vordergrund aktiv.

Das Editorfenster zeigt zusätzlich die aktuelle Zeilen- und Spaltennummern in der unteren linken Ecke an. Wurde die Datei geändert, erkennen Sie dies an einem * links neben den Spalten- und Zeilennummern.

Und so sieht ein typisches Fenster aus:

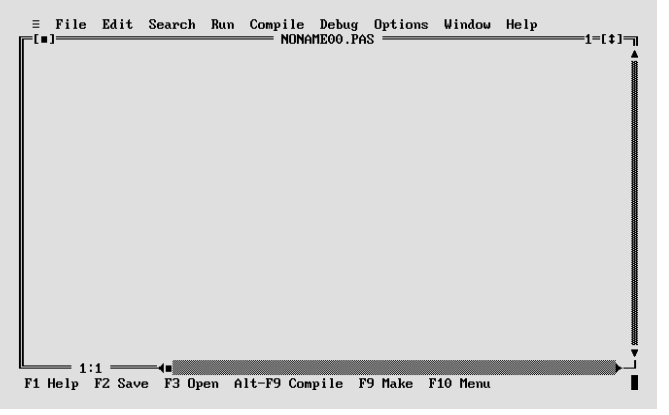


Abbildung 1.1 Ein typisches Fenster

Das *Schließfeld* finden Sie in der oberen linken Ecke. Durch Anklicken bzw. mit *Window/Close* oder mit Alt F3 können Sie ein Fenster schließen. Ein Sonderfall ist das Hilfe-Fenster, denn dieses gilt als temporäres Fenster und wird mit ESC geschlossen.

Der *Titel*, der oberste horizontale Balken, enthält den Namen des Fensters und die Fensternummer. Durch doppeltes Anklicken können Sie das Fenster auf volle Größe vergrößern bzw. die ursprüngliche Größe wiederherstellen, durch ein einfaches Anklicken können Sie das Fenster verschieben.

Jedes Fenster, das Sie in Turbo Pascal öffnen, hat eine *Fensternummer* im oberen rechten Rand. Mit Alt 0 erhalten Sie eine Liste der momentan geöffneten Fenster. Durch die Tastenkombination Alt Fensternummer können Sie gezielt ein bestimmtes Fenster aktivieren. Zum Beispiel: das Hilfe-Fenster habe die Nummer 5 und wird von anderen Fenstern verdeckt, mit Alt 5 können Sie es jederzeit aktivieren und damit in den Vordergrund bringen.

Das *Zoom-Feld* eines Fensters finden Sie in der oberen rechten Ecke. Wird in

dieser Ecke als Sinnbild ein Pfeil nach oben (ï) angezeigt, können Sie den Pfeil anklicken und das Fenster bis zu seiner maximalen Ausdehnung vergrößern. Ein Pfeil mit zwei Spitzen als Sinnbild zeigt an, daß das Fenster bereits in seiner maximalen Größe dargestellt wird. In diesem Fall wird das Fenster nach dem Anklicken dieses Pfeiles auf seine ursprüngliche Größe zurückgesetzt. Wenn Sie die Tastatur benutzen, dann vergrößern oder verkleinern Sie ein Fenster mit *Window/Zoom* oder durch Betätigen der F5-Taste.

Rollbalken sind horizontale oder vertikale Balken am rechten bzw. unteren Fensterrand, die an beiden Enden durch einem Pfeil begrenzt sind. Sie enthalten eine Positionsmarke, die die Position des Fensterausschnittes relativ zum Gesamttext anzeigt. Einen Rollbalken können Sie mit der Maus benutzen, um den Inhalt eines Fensters zu »rollen«. Klicken Sie an einem beliebigen Ende des Balkens das Pfeil-Symbol an, so wird immer eine Zeile bzw. Spalte gerollt. Kontinuierliches Rollen erfolgt, wenn Sie die Maustaste gedrückt halten und die Positionsmarke ziehen. Möchten Sie jeweils genau eine Seite rollen, klicken Sie die schattierten Bereiche an einer Seite des Rollbalkens an. Außerdem können Sie die Positionsmarke zu einem beliebigen Punkt auf dem Rollbalken ziehen, wenn Sie schnell an eine andere Position innerhalb des Fensters, bezogen auf die augenblickliche Stellung der Positionsmarke, gelangen möchten.

Das Feld zum *Verändern der Größe* finden Sie in der unteren rechten Ecke (an dieser Stelle ist der Fensterrahmen nur mit einer einfachen Linie gezeichnet). Bewegen Sie diese Ecke, so wird das Fenster größer bzw. kleiner. Über die Tastatur ändern Sie die Größe mit *Size/Move* aus dem Window-Menü oder mit Ctrl F5 .

Verwaltung der Fenster

In Tabelle 1.7 finden Sie einen Kurzüberblick über die Fenster in Turbo Pascal. Sie können alle Aktionen sowohl mit der Maus als auch mit der Tastatur durchführen.

Tabelle 1.7 Fenster manipulieren

Ergebnis	Arbeitsmethoden
Öffnen eines Editorfensters	Wählen Sie *File/Open* oder drücken Sie F3 .
Öffnen anderer Fenster	Wählen Sie das gewünschte Fenster aus dem Window-Menü.
Schließen eines Fensters	Wählen Sie im Window-Menü *Close*, drücken Sie Alt F3 oder klicken Sie das Schließfeld an.
Aktivieren eines Fensters	Klicken Sie das Fenster an einer beliebigen Stelle an, oder drücken Sie Alt und die Fensternummer, angezeigt im oberen rechten Eck oder wählen Sie *Window/List* oder Alt 0 und wählen Sie das Fenster aus der Liste aus oder wählen Sie *Window/Next* oder F6 und aktivieren Sie damit das nächste Fenster.
Verschieben des aktiven Fensters	Verschieben Sie den Titelbalken oder drücken Sie Ctrl F5 (*Window/Size/Move*) und verwenden Sie die Pfeiltasten zum Positionieren des Fensters. Drücken Sie anschließend Enter .

Ergebnis	Arbeitsmethoden
Verkleinern des aktiven Fensters	Verschieben Sie die Ecke zum Verkleinern (oder auch eine beliebige andere) oder wählen Sie *Window/Size/Move* und drücken Sie Shift , solange Sie die Pfeiltasten verwenden. Drücken Sie dann Enter . Wollen Sie ein Tastenkürzel verwenden, drücken Sie Ctrl F5 und arbeiten dann mit Shift und den Pfeiltasten.
Vergrößern des aktiven Fensters	Klicken Sie das Zoom-Feld im oberen rechten Eck an oder klicken Sie den Titel zweimal an oder wählen Sie *Window/Zoom* oder drücken Sie F5 .

Die Statuszeile

Die Statuszeile befindet sich in der untersten Zeile des Turbo-Pascal-Bildschirms. Die Funktionen der Statuszeile:

- Sie zeigt die aktuell verfügbaren Tastenkürzel für die Menü-Befehle, die Sie, anstatt sie im Menü zu wählen, auch in der Statuszeile anklicken bzw. eingeben können.
- Sie meldet, was das Programm gerade macht. Beim Sichern einer Datei wird beispielsweise `Saving Dateiname...` ausgegeben.
- Sie finden einzeilige Hinweise auf den gewählten Menü-Befehl und die Wahlpunkte des Dialogfensters.

Die Statuszeile ändert sich, wenn Sie die Fenster wechseln oder bestimmte Menüs-Befehle aufrufen.

Beim Editieren und Schreiben von Programmen im Editorfenster sieht die Statuszeile folgendermaßen aus:

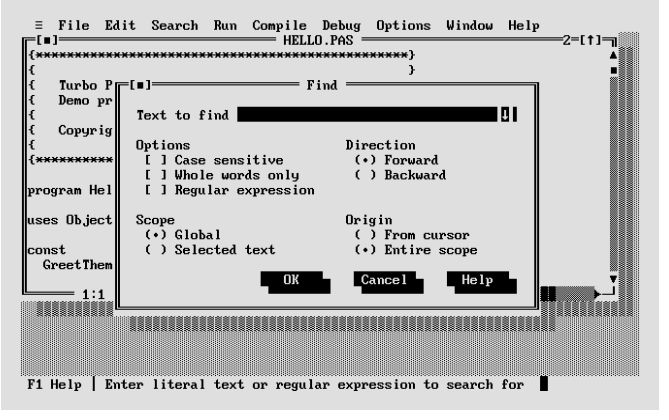


Abbildung 1.2 Eine typische Statuszeile

Dialog-Fenster

Menü-Befehle mit Fortsetzungspunkten (...) öffnen ein Dialogfenster mit mehreren Optionen und Einstellungen.

Für Einstellungen in Dialogfenstern stehen Ihnen fünf verschiedene Bildschirm-Steuerungen zur Verfügung: Schaltfelder, Markierungsfelder, Aktionsschalter, Eingabefelder und Listen. Bei Farbmonitoren werden die einzelnen Elemente des Dialogfensters in verschiedenen Farben dargestellt. Hier ein typisches Dialogfenster:

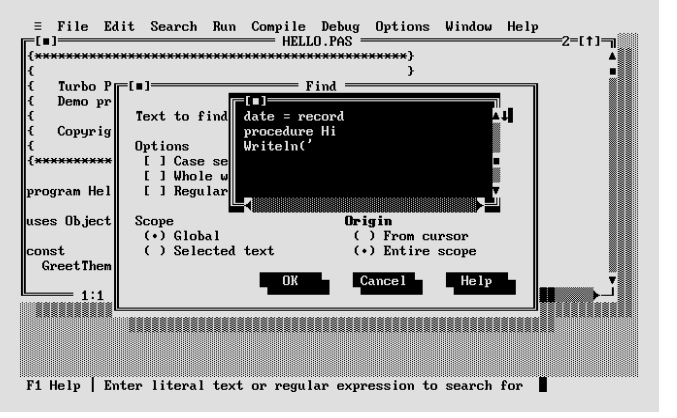


Abbildung 1.3
Ein typisches Dialog-Fenster

In diesem Dialogfenster finden Sie drei *Aktionsschalter*: *OK*, *Cancel* und *Help*. Der Schalter *OK* bewirkt die Umsetzung der getroffenen Auswahl in Turbo Pascal. Mit *Cancel* verlassen Sie das Dialogfenster wieder und verwerfen die Eingaben. *Help* öffnet ein Hilfefenster zu diesem Dialogfenster. Esc ist ein Tastenkürzel für *Cancel* (auch wenn kein *Cancel*-Aktionsschalter erscheint).

Die Betätigung der Aktionsschalter erfolgt durch Anklicken mit der Maus oder über die Tastatur durch Drücken der Alt-Taste und dem hervorgehobenen Buchstaben: Mit Alt K betätigen Sie z.B. den *OK*-Aktionsschalter. Die Tasten Tab (vorwärts) bzw. Shift Tab (rückwärts) dienen innerhalb eines Dialogfensters zur Aktivierung der einzelnen Elemente. Aktivierte Elemente werden immer hervorgehoben dargestellt. Mit Tab springen Sie zum nächsten Aktionsschalter; mit Enter wird er betätigt.

Im Dialogfenster der Abbildung 1.3 ist die Voreinstellung *OK*, d.h. Sie wählen den Aktionsschalter durch Drücken von Enter . Bei Monochrom-Systemen wird die Voreinstellung mit Pfeilen, bei Farbmonitoren hervorgehoben angezeigt. Beachten Sie, daß durch die Anwahl eines Aktionsschalters mit Tab die Voreinstellung geändert wird.

Markierungsfelder und Schaltfelder

Gewählte *Markierungsfelder* werden durch ein *x* gekennzeichnet. Ein leeres Feld bedeutet also, daß die Option ausgeschaltet ist (*OFF*). Ein Markierungsfeld

wird durch Anklicken des Feldes bzw. des zugehörigen Textes gewählt, oder indem Sie Tab (bis das Markierungsfeld hervorgehoben wird) drücken und anschließend Leertaste drücken oder indem Sie Alt und den hervorgehobenen Buchstaben eingeben.

Sie können beliebig viele Markierungsfelder auswählen. Markierungsfelder werden manchmal auch zu einer Gruppe zusammengefaßt. Ist die Gruppe mit Tab angewählt, markieren Sie mit den Cursortasten das gewünschte Element und schalten es mit der Leertaste ein bzw. aus. Bei Monochrom-Bildschirmen werden die aktiven Markierungsfelder bzw. Gruppen von Markierungsfeldern mit einem Chevron-Symbol, (») dargestellt. Nach Drücken von Tab springt das Symbol zur nächsten Gruppe von Markierungsfeldern oder Schaltfeldern.

Schaltfelder sind im Gegensatz zu Markierungsfeldern prinzipiell *immer* in Gruppen zusammengefaßt. Der entscheidende Unterschied zu Makierungsfeldern besteht darin, daß die Wahl *eines* Schaltfeldes die Wahl eines anderen Schaltfeldes innerhalb der gleichen Gruppe ausschließt. Die Wahl erfolgt mit der Maus durch Anklicken des Schaltfeldes bzw. des zugehörigen Textes, oder mit der Tastatur durch Drücken der Alt-Taste und des hervorgehobenen Buchstabens. Sie können auch Tab drücken, bis die Gruppe hervorgehoben wird, und dann mit den Cursortasten das Schaltfeld anwählen. Mit Tab oder Shift Tab verlassen Sie die Gruppe mit dem neu gewählten Schaltfeld wieder.

Eingabefelder und Listen

Dialogfenster können auch Eingabefelder enthalten. Die meisten Editier-Tasten haben auch in diesem Fenstertyp ihre gewohnte Funktion (z.B. die Cursortasten, Home , End und Ins). Sollten Sie beim Eintippen einmal bis ganz an den Rand des Eingabefeldes gelangen, dann wird der Fensterinhalt automatisch weitergerollt. Außerdem erscheinen, falls das Eingabefeld mehr Text beinhaltet als im dafür reservierten Bildschirmbereich dargestellt werden kann, an den Enden des Fensters Pfeilspitzen als Symbole, die zum Rollen und Verschieben des Textes mit der Maus dienen.

Ein ↓-Symbol auf der rechten Seite des Eingabefeldes gibt an, daß zu diesem Feld eine Eingabeaufzeichnungsliste existiert, welche durch Anklicken des ↓-Symbols oder über die Taste → geöffnet wird. In der Liste finden Sie die letzten von Ihnen eingebenen und verwendeten Texte. Die Aufzeichnungsliste des *Find*-Feldes zeigt z.B. Begriffe, die Sie bereits gesucht haben. Wählen Sie einen der alten Einträge, editieren Sie diese Zeile nötigenfalls, und bestätigen Sie Ihre Wahl mit Enter . Mit Esc brechen Sie die Funktion ab und verlassen die Eingabeaufzeichnungsliste.

Ein weiterer Bestandteil vieler Dialogfenster ist eine Liste mit variabler Länge. Sie können durch die Liste rollen und Elemente auswählen, ohne dabei das Dialogfenster zu verlassen. Erscheint in einer Liste ein blinkender Cursor, zeigt dies an, daß eine Hilfsfunktion für das Suchen von Begriffen aktiviert ist. Wenn Sie also nach einem bestimmten Begriff suchen, so brauchen Sie ihn bzw. seine ersten Buchstaben nur noch einzutippen, und Turbo Pascal sucht nach diesem Begriff.

Ein Listen-Fenster aktivieren Sie mit dem hervorgehobenen Buchstaben aus dem Titel des Listen-Fensters, durch Anklicken oder durch Drücken der Tab -

Taste, solange bis das Fenster hervorgehoben ist. Den Inhalt der Liste bewegen Sie mit Hilfe der Positionsmarke oder mit den Cursortasten.

Neue Eigenschaften des integrierten Editors

Wenn Sie schon länger mit Borland-Produkten arbeiten, erfahren Sie in nachstehender Zusammenfassung, was sich geändert hat. Der integrierte Editor von Turbo Pascal verfügt jetzt über:

- Mausunterstützung
- Unterstützung von großen Dateien (bis zu 1 MByte; die Obergrenze liegt bei 8 MByte für alle Editor-Fenster zusammen)
- Die Tasten Shift ← → ↓ ↑ zum Markieren von Text
- Editor-Fenster, deren Größe veränderbar ist, die sich überlappen können und die Sie auch verschieben können
- Multidatei-Fähigkeiten, wodurch Sie mehrere Dateien gleichzeitig öffnen können
- Mehrere Fenster, so daß Sie verschiedene Stellen einer Datei oder verschiedene Dateien zugleich am Bildschirm sehen können
- Eine hochentwickelte Makro-Sprache zur Erstellung eigener Editor-Kommandos (siehe die Datei TEMC.DOC auf Ihren Turbo-Pascal-Disketten)
- Die Möglichkeit, Text oder Beispiele aus dem Hilfe-Fenster einzufügen
- Eine editierbare Zwischenablage, mit deren Hilfe Sie Text ausschneiden, kopieren und von einem Fenster in ein anderes kopieren können

Der Start der integrierten Entwicklungsumgebung

Wenn Sie mit einem Diskettenlaufwerk arbeiten: Legen Sie die Turbo-Pascal-Systemdiskette in das Laufwerk A ein und tippen Sie nach dem DOS-Prompt folgenden Befehl ein:

```
turbo Enter
```

Nachdem der Startbildschirm von Turbo Pascal erschienen ist, legen Sie die Arbeitsdiskette in das Laufwerk ein.

Hinweis: Sollte Ihr Rechner nur mit zwei 5 1/4 Zoll-Laufwerken mit niedriger Schreibdichte ausgerüstet sein, dann müssen Sie Turbo Pascal für den Betrieb mit zwei Laufwerken konfigurieren.

Wenn Sie mit einer Festplatte arbeiten: Starten Sie den Computer wie gewohnt, und geben Sie nach Erscheinen des DOS-Prompt die folgenden Befehle ein:

```
cd \tp Enter
turbo Enter
```

Hinweis: Der Name einer Quelltext-Datei läßt sich auch als Kommandozeilen-Parameter angeben. Der Aufruf von Turbo Pascal mit dem Befehl

```
turbo nummer1
```

hat dieselbe Wirkung wie die Auswahl von *File/Open* und die Angabe von NUMMER1 unmittelbar nach dem Start: Wenn die Datei NUMMER1.PAS existiert, wird sie in den Editor geladen; existiert sie nicht, dann gibt Turbo Pascal diesen Namen anstelle von NONAME.PAS vor. In beiden Fällen wird der Editor automatisch aktiviert.

Die Eingabe des Quelltextes

Nachdem Sie Turbo Pascal aufgerufen haben, sehen Sie auf dem Startbildschirm das Hauptmenü, die jetzt noch leere Arbeitsfläche, eine Statuszeile und ein Fenster mit Informationen über Ihre Turbo-Pascal-Version (dieses Informations-Fenster erhalten Sie auch über den Menü-Befehl *About* aus dem System-Menü). Drücken Sie eine beliebige Taste, um das Informations-Fenster zu schließen.

Drücken Sie nun F10, um das das Hauptmenü zu aktivieren, und danach F3 zum Öffnen des Dialogfensters *File/Open*. Der Cursor befindet sich nun in dem Eigabefeld für den Dateinamen. Geben Sie hier `Nummer1` ein und drücken Sie anschließend die Enter -Taste (die Extension .PAS brauchen Sie nicht einzugeben, da sie von Turbo Pascal automatisch hinzugefügt wird). Nun können Sie damit beginnen, Ihr erstes Programm zu schreiben:

```
program Nummer1;
begin
  Writeln('Hallo!');
end.
```

Zuerst einmal ein ganzes Bündel von Hinweisen zu diesen Zeilen:

- »Reservierte Wörter« wie **program** sind in diesem Handbuch fettgedruckt, um die Listings übersichtlicher zu gestalten – eine spezielle Art der Eingabe gibt es für sie nicht, sie erscheinen innerhalb des *Edit*-Fensters genauso wie der restliche Text.

- Jede dieser vier Programmzeilen wird mit Enter abgeschlossen.

- Für die Korrektur von Tippfehlern stehen Ihnen die Cursortasten ↑, ↓, ← und → zur Verfügung, Backspace löscht jeweils das Zeichen, das sich links des Cursors befindet. Korrekturen sind auch dann noch möglich, wenn die betreffende Zeile bereits mit Enter abgeschlossen wurde. (Die Editor-Kommandos werden in Kapitel 8 »Der Editor von A bis Z« besprochen.)

Überprüfen Sie Ihre Eingaben noch einmal genau (die erste und die dritte Zeile enden mit einem Semikolon, die letzte Zeile endet mit einem Punkt).

Die Compilierung

Ein Druck auf F10 beendet den Editor und bringt Sie zur Menüleiste von Turbo Pascal zurück. Aktivieren Sie nun mit C oder Alt C das Menü *Compile* und wählen Sie mit einem weiteren Druck auf C den Punkt *Compile* innerhalb

dieses Menüs (die Kombination Alt F9 ist ein Tastenkürzel für diese Operation).

Wenn Sie keinen Tippfehler gemacht haben, erscheint ein Fenster des Compilers auf dem Bildschirm, in dessen unterster Zeile die Meldung `Compile success: Press any key` blinkt. Falls dem nicht so ist, finden Sie sich automatisch im Editor wieder – die oberste Zeile des Fensters *Edit* zeigt eine Fehlermeldung, der Cursor befindet sich an der Stelle, die der Compiler als fehlerhaft moniert. Korrigieren Sie Ihren Text entsprechend, beenden Sie den Editor mit F10 und rufen Sie den Compiler (mit Alt C C) erneut auf.

Der Start

Nachdem Sie die Erfolgsmeldung des Compilers erhalten haben, drücken Sie irgendeine Taste: Das kleine Fenster verschwindet wieder vom Bildschirm, das Editor-Fenster ist wieder aktiviert. Ein Druck auf F10 bringt Sie zur Menüleiste zurück.

Aktivieren Sie das Menü *Run* (mit Alt R) und wählen Sie in diesem Menü den Punkt *Run* (oder geben Sie einfach den Tastenbefehl Ctrl F9).

Haben Sie's gesehen? Wahrscheinlich nicht so ganz – eher ein kurzes Flackern des Bildschirms: Turbo Pascal hat auf den »DOS-Bildschirm« umgeschaltet, NUMMER1 ausgeführt und Ihnen dann sofort wieder die Menüleiste und das Fenster des Editors präsentiert.

Wählen Sie *Window/User screen* (oder geben Sie einfach den Tastenbefehl Alt F5). Turbo Pascal schaltet nun permanent auf den »DOS-Bildschirm« um, und Sie können das Ergebnis Ihres ersten Programms in aller Ruhe bewundern. Es sollte in etwa so aussehen:

```
Hallo!
```

Ein Druck auf eine beliebige Taste bringt Sie zur integrierten Entwicklungsumgebung zurück.

Die Speicherung des Quelltextes

Damit unser Meisterwerk nicht verlorengeht, wenn Sie den Computer irgendwann einmal abschalten, muß es als Datei gespeichert werden: Drücken Sie F2 oder wählen Sie *Save* aus dem *File*-Menü.

Zur Funktion eines Compilers

Das Herz des IBM PC ist – wie bei den meisten Mikrocomputern – ein Mikroprozessor (CPU), der den größten Teil der Arbeitslast trägt. Im Falle der DOS-Maschinen stammt dieser Chip von der Firma Intel und kommt aus der Prozessor-Familie iAPx86 – ob es sich in Ihrem Computer dabei um einen 8088, 8086, 80186, 80286, 80386 oder gar einen 80486 handelt, spielt keine Rolle: Turbo Pascal benutzt nur Befehle, die von allen diesen Prozessor-Versionen verstanden werden.

Apropos »Verstehen«: Das einzige, was ein Prozessor versteht, sind binär

codierte Befehle. Sie veranlassen ihn dazu, bestimmte Operationen auszuführen, wie beispielsweise das Zeichnen von Objekten auf dem Bildschirm, die Berechnung mathematischer Funktionen oder die Manipulation von Text und Daten. Kurz gesagt: er macht alles, was man von ihm will, wenn man ihm nur die richtigen Befehlsfolgen gibt. Instruktionen auf dieser Ebene werden als *Maschinencode* bezeichnet.

Weil der Maschinencode nur aus binärer Information besteht, ist er für Menschen weder leicht zu schreiben noch zu verstehen. Mit einem Assembler-Programm lassen sich Befehlsfolgen auf Maschinenebene in lesbarer Form erstellen – allerdings müssen Sie dazu genau wissen, wie der Prozessor intern arbeitet. Außerdem werden Sie feststellen, daß vergleichsweise einfache Operationen wie z.B. die Ausgabe einer Zahl eine Unmenge einzelner Befehle erfordern.

Wer sich nicht mit Assemblerprogrammierung und Maschinensprache herumschlagen will, benutzt eine Hochsprache wie Pascal. Pascal-Programme sind leicht zu schreiben und zu lesen, weil diese Sprache nicht für Prozessoren, sondern für Menschen geschrieben wurde. Es bleibt die Tatsache, daß der Prozessor des PC nur binären Code versteht. Der Turbo-Pascal-Compiler übersetzt (compiliert) deshalb das von Ihnen als Quelltext eingegebene Pascal-Programm in binäre Befehlsfolgen. Auch ein Compiler ist nichts weiter als ein Programm zur Manipulation von Daten: Er liest einen Quelltext ein und liefert eine Folge von Maschinenbefehlen zurück.

Wo wird ein übersetztes Programm gespeichert?

Wenn Sie ein Programm compilieren lassen, speichert Turbo Pascal den resultierenden Maschinencode im Hauptspeicher (RAM) des Computers. Das hat einige Vorteile:

- Der Compiler arbeitet schneller – es dauert wesentlich länger, auf eine Diskette oder Festplatte zu schreiben.

- Weil sich das übersetzte Programm bereits im RAM befindet, muß es zu seiner Ausführung nicht erst geladen werden. Turbo Pascal teilt dem Betriebssystem lediglich mit, daß das Programm ausgeführt werden soll.

- Turbo Pascal befindet sich ebenfalls im RAM und kann deswegen nach Ausführung des Programms sehr schnell wieder gestartet werden.

Nachdem diese Methode so hervorragend funktioniert, stellt sich natürlich die Frage, warum man es nicht immer so macht?

Zwei Gründe sprechen dagegen: Erstens können Sie Ihre Programme auf diese Art *nur* von Turbo Pascal aus starten. Innerhalb des RAM compilierte Programme können weder auf die Diskette gespeichert noch von der DOS-Kommandoebene aus gestartet oder kopiert werden.

Zweitens – wenn auch etwas weniger wahrscheinlich – können sich Probleme mit der Speicherkapazität des Computers ergeben. Falls Sie einen »Kompatiblen« mit 512 KByte RAM Ihr eigen nennen, möglichst noch einige Zusatz-

programme wie SideKick geladen haben und dann ein Programm schreiben, das viel mit dynamischer Datenverwaltung arbeitet, dürften Sie in Schwierigkeiten kommen.

Die Erzeugung einer .EXE-Datei (d.h. eines Programms, das unabhängig von der integrierten Entwicklungsumgebung gestartet werden kann) ist mehr als einfach: Aktivieren Sie das Menü *Compile* und wählen Sie den Befehl *Destination*, der dadurch von *Memory* (Speicher) auf *Disk* umschaltet. Folgende Aufrufe des Compilers legen den erzeugten Code nun nicht mehr im Hauptspeicher des Computers, sondern als Datei auf der Diskette ab. Auf diese Weise erzeugte Dateien haben denselben Namen wie der Quelltext, aber die Extension .EXE – die Compilierung eines Programms NUMMER1.PAS mit dem »Ziel« *Disk* erzeugt also eine Datei namens NUMMER1.EXE.

Die Auswahl von *Run/Run* startet Ihr Programm übrigens auch dann, wenn es als .EXE-Datei auf die Diskette compiliert wurde – Turbo Pascal merkt von selbst, ob das Programm direkt aus dem Hauptspeicher gestartet werden kann oder ob es zuerst geladen werden muß.

Das zweite Programm

Programme, die nur einen vordefinierten Text ausgeben, sind auf die Dauer recht langweilig – deshalb folgt hier eine *interaktive* Version von NUMMER1:

```
program Nummer2;
var
  Name : string;
begin
  Write('Geben Sie Ihren Namen ein: ');
  Readln(Name);
  Writeln('Hallo ', Name, '!');
end.
```

Wenn Sie zur Eingabe dieses Textes den Editor (mit E von der Menüleiste aus oder durch den Tastenbefehl Alt E) aktivieren, dann werden Sie feststellen, daß hier nach wie vor unser erstes Programm gespeichert ist. An diesem Punkt stehen Ihnen gleich drei Möglichkeiten zur Verfügung:

- Sie verändern den Text des ersten Programms. Danach sollten Sie allerdings *nicht* F2 zur Speicherung verwenden – verlassen Sie stattdessen den Editor nach der Eingabe mit F10 , aktivieren Sie das Menü *File* und wählen Sie dort *Save as*. Geben Sie in das Eingabefeld des daraufhin geöffneten Dialogfensters *Nummer2* ein, und schließen Sie die Eingabe mit Enter ab. (Wie der Titel des Editor-Fensters danach zeigt, verwendet Turbo Pascal den von Ihnen angegebenen Namen nun für alle weiteren Operationen – weitere Speicherungen des Textes sind also auch über F2 möglich).

- Innerhalb des Editors drücken Sie F3 (für *File/Open*) und beantworten die daraufhin erscheinende Anfrage mit NUMMER2 und Enter . Turbo Pascal stellt

fest, daß eine Datei dieses Namens noch nicht existiert und legt eine neue Datei an – der alte Inhalt des Fensters *Edit* wird gelöscht, und Sie können den neuen Text eingeben, der durch einen Druck auf F2 als NUMMER2.PAS gespeichert wird.

■ Sie verlassen den Editor, aktivieren das Menü *File*, wählen dort *Open* und geben den Dateinamen NUMMER2 an. Die Wirkung ist dieselbe wie der Tastenbefehl F3 innerhalb des Editors.

Hinweis: Falls Sie unser erstes Programm noch nicht gespeichert oder seit der letzten Speicherung noch einmal verändert haben, wird es nicht kommentarlos aus dem Editor gelöscht – Sie erhalten zuerst eine entsprechende Rückfrage.

Nach der Eingabe des Textes verlassen Sie den Editor mit F10, aktivieren das Menü *Run* aus der Menüleiste und wählen dort *Run*, um das Programm zu compilieren und direkt danach zu starten. Falls der Compiler Fehler entdeckt, ist der Ablauf auch bei *Run/Run* derselbe wie bei unserem ersten Programm: Turbo Pascal aktiviert den Editor, gibt eine entsprechende Fehlermeldung aus und erwartet eine Korrektur des Textes.

Wenn alles so läuft wie geplant und Sie sich über die freundliche Begrüßung gefreut haben (zu deren Bewunderung Sie wieder erst mit Alt F5 auf den »DOS-Bildschirm« umschalten müssen), können wir daran gehen, uns ein wenig mit den Einzelheiten auseinanderzusetzen:

Die erste Zeile von *Nummer2* legt den Namen des Programms fest. Dieser Name hat im Moment keine weitere Bedeutung, weil wir nur mit einem einzigen Programmteil arbeiten. Die zweite und dritte Zeile enthalten eine *Variablen-Deklaration*: Wir teilen dem Compiler mit, daß Platz für die Speicherung einer Zeichenkette (*string*) reserviert werden soll, und geben diesem »Platz« den Namen *Name*. *Name* ist ein Symbol: *Wo* im Hauptspeicher des Computers dieser Platz reserviert wird, ist nicht weiter von Interesse – wichtig ist nur, daß wir diesen Speicherbereich nach der Deklaration über den zuvor vereinbarten Namen ansprechen können. Das Wort `begin` in der vierten Zeile kennzeichnet den Start des eigentlichen Programms: Der Compiler erkennt daran, daß nun keine weiteren Variablen-Deklarationen folgen, sondern auszuführende *Anweisungen*.

Gehen wir davon aus, daß die *Write*-Anweisung in der fünften Zeile selbsterklärend ist und beschäftigen wir uns mit *Readln*(*Name*): Hier wird eine Eingabe über die Tastatur angefordert, die mit Enter abgeschlossen werden muß und in der Variablen *Name* gespeichert wird. Die *Writeln*-Anweisung in der vorletzten Zeile demonstriert, daß sich Variablen wie *Name* nicht nur einlesen, sondern auch wieder ausgeben lassen: Auf dem Bildschirm erscheint der von Ihnen eingegebene Text zusammen mit einem vorangestellten `Hallo` und einem nachfolgenden Ausrufezeichen. Das **end** in der letzten Zeile hat für die Anweisungen eine ähnliche Funktion wie zuvor das **begin** für die Variablen-Deklaration: es teilt dem Compiler mit, daß dieser Teil (und damit das Programm) abgeschlossen ist.

Das dritte Programm

ist ebenfalls interaktiv: Es fordert zwei Eingaben an, nimmt eine Berechnung vor und präsentiert schließlich das Ergebnis. Hier ist es:

```
program Nummer3;
var
  A, B, C: Integer;
begin
  Write('Geben Sie zwei Zahlen ein: ');
  Readln(A, B);
  C := A div B;
  Writeln('Das Ergebnis ist: ', C);
end.
```

Wir haben drei Variablen (*A*, *B* und *C*) deklariert, die allerdings nicht Zeichenfolgen wie *Name*, sondern ganzzahlige Werte (*Integer*) speichern. Nachdem *Write* die Aufforderung `Geben Sie zwei Zahlen ein:` ausgegeben hat, erwartet *Readln* zwei ganzzahlige Werte, die Sie beispielsweise so eingeben können:

```
35 m  Enter
```

Das Programm nimmt eine Division des ersten eingegebenen Wertes durch den zweiten vor und liefert schließlich das Ergebnis (das Sie sich – wie nun schon gewohnt – mit Alt F5 auf den Bildschirm holen müssen):

```
Das Ergebnis ist: 5
```

Was passiert, wenn Sie anstelle von numerischen Werten Buchstaben eingeben? Starten Sie *Nummer3* erneut und probieren Sie es aus:

```
35 7  Enter
```

Sofort erscheint wieder das Editorfenster, der Cursor steht auf der Zeile mit der *Readln*-Anweisung, und in der obersten Zeile des Fensters erscheint die Meldung

```
Runtime error 106 at 0000:0038
```

versehen mit dem Hinweis auf ein »illegales numerisches Format«. Wir werden uns am Ende von Kapitel 2 »Die sieben Elemente der Programmierung« noch eingehender mit derartigen Widrigkeiten beschäftigen.

Grafische Spielereien

Unser viertes Beispiel soll zumindest ein wenig andeuten, was in Turbo Pascal eigentlich steckt – es setzt allerdings voraus, daß Ihr Computer mit einem Grafik-Adapter ausgerüstet ist. Falls Sie nicht so genau wissen, ob das der Fall ist, probieren Sie es am besten einfach aus. Schlimmstenfalls erhalten Sie eine Fehlermeldung – ein doppelter Druck auf Enter sollte Sie in jedem Fall zur integrierten Entwicklungsumgebung zurückbringen.

Das Listing unseres vierten Programms ist im Vergleich zu den vorhergehenden Beispielen recht umfangreich – um seine Details kümmern wir uns in den nächsten drei Kapiteln.

```
program Moire;
uses  Graph;
const
  Start   = 25;
  Ende    = 175;
  Abstand = 2;

var
  GraphDriver : Integer;     { Kenn-Nummer des Grafiktreibers }
  GraphMode   : Integer;     { Grafikmodus }
  ErrorCode   : Word;        { speichert Fehlernummern }
  X1,Y1,X2,Y2 : Integer;

begin
  GraphDriver := Detect;    { automatische Erkennung }
  InitGraph(GraphDriver, GraphMode, '\TP');
  ErrorCode := GraphResult;
  if ErrorCode <> 0 then    { Fehler ? }
    begin
      Writeln('Grafik-Fehler: ', GraphErrorMsg(ErrorCode));
      Writeln('(Wahrscheinlich ist Ihr Computer nicht ' +
                `grafikfähig.)');
      Writeln('Programm abgebrochen...');
      Halt(1);
    end;
  Y1:= Start; Y2 := Ende; X1:= Start;
  while X1 <= Ende do
  begin
    X2:= (Start+Ende) - X1;
    Line(X1,Y1,X2,Y2);
    X1:= X1 + Abstand;
   end;
  X1:= Start; X2:= Ende; Y1:= Start;
  while Y1 <= Ende do
  begin
    Y2:= (Start+Ende) - Y1;
```

```
    Line(X1,Y1,X2,Y2);
    Y1 := Y1 + Abstand;
   end;

  OutText('Das war''s. Weiter mit <RETURN>...');
  Readln;
  CloseGraph;
end.
```

Hinweis: Das Programm setzt voraus, daß sich die Unit GRAPH.TPU und ein zu ihrem Video-Adapter passender *Grafik-Treiber* (.BGI-Datei) im aktuellen Verzeichnis befinden. (Wenn Sie mit Diskettenlaufwerken arbeiten: Hinweise zum »Auspacken« der Datei BGI.ZIP finden Sie im Abschnitt »Installation« der Einführung dieses Handbuches).

Wenn Sie eine deutsche Tastatur verwenden, können Sie die geschweiften Klammern entweder über Alt erzeugen ({ = Alt 1 2 3 , } = Alt 1 2 5) oder durch (* und *) ersetzen.

Wenn der Compiler Ihre Version dieses Programms fehlerfrei übersetzt, speichern Sie den Text mit F2 und starten das Programm mit *Run*. Es erzeugt ein Quadrat mit weißem Zentrum und recht interessanten Mustern an den Rändern, die als Moiré-Muster bekannt sind.

Einige Einzelheiten: Die Anweisung **uses** teilt dem Compiler mit, daß eine *Unit* namens *Graph* in das Programm aufgenommen werden soll. Eine *Unit* ist eine Art Bibliothek – eine Sammlung von Prozeduren, Funktionen und Daten-Deklarationen, die dem Programm über **uses** zur Verfügung gestellt werden. In unserem Beispiel machen wir von den in *Graph* definierten Prozeduren *InitGraph*, *Line* und *CloseGraph* sowie den Funktionen *GraphResult* und *GraphErrorMsg* Gebrauch.

Der mit dem reservierten Wort **const** eingeleitete Programmteil definiert drei numerische Konstanten (*Start*, *Ende* und *Abstand*), die Größe, Position und Aussehen des Quadrats bestimmen. Wenn Sie Lust haben, können Sie mit diesen drei Werten ausgiebig experimentieren.

Vorsicht: Der Wert von *Abstand* darf nicht kleiner als 1 (d.h. auf 0 oder negativ) gesetzt werden! Ansonsten wird der Endwert der Schleife nie erreicht und das Programm läuft endlos – der einzige Ausweg ist dann ein Neustart des Computers.

Die Variablen *X1*, *Y1*, *X2* und *Y2* stehen für Positionen auf der jeweils entgegengesetzten Seite des Quadrats. Das Quadrat selber entsteht folgendermaßen: Vom Punkt (*X1,Y1*) wird eine Linie nach (*X2,Y2*) gezeichnet. Danach werden Anfangs- und Endpunkt um den Wert von *Abstand* verändert, der Kreislauf wiederholt sich. Die erste Linie innerhalb der Schleife ist die Diagonale des Quadrats von der linken oberen zur rechten unteren Ecke (vom Punkt (25,25) zum Punkt (175,175)), die nächste geht von (27,25) zu (173,25)..., die letzte ist die Diagonale von der rechten oberen in die linke untere Ecke, geht also vom Punkt (175,25) zum Punkt (25,175).

Die zweite Schleife des Programms ist der ersten sehr ähnlich. Hier werden anstelle der X-Koordinaten die Y-Werte verändert, also die Anfangs- und Endpunkte der Linien vertikal bewegt.

Die *Readln*-Anweisung am Ende des Programms sorgt dafür, daß das Zurück-

schalten in den Textmodus erst nach einem Druck auf Enter stattfindet – ohne sie würde das gezeichnete Bild sofort nach seiner Fertigstellung vom Bildschirm verschwinden.

Hinweis: Eine nachfolgende Umschaltung auf den »DOS-Bildschirm« mit Alt F5 hat hier nicht den gewünschten Effekt – dieser Bildschirm speichert nur Textausgaben.

2. Die sieben Elemente der Programmierung

Ab jetzt wird es ernsthaft. Deshalb beginnen wir (anstelle der üblich launigen Einleitung oder einem geschichtlichen Rückblick) dieses Kapitel mit einer formalen Definition: *Ein Programm löst gestellte Aufgaben durch Manipulation von Daten.* Um ein beliebiges Problem zu lösen, muß es also

- Daten von irgendwoher erhalten;
- Platz zur Speicherung dieser Daten zur Verfügung stellen;
- Anweisungen zur Bearbeitung der Daten enthalten;
- die Ergebnisse dieser Bearbeitung an den Benutzer ausgeben (mit letzterem sind normalerweise Sie gemeint).

Die Anweisungen zur Bearbeitung von Daten lassen sich auf drei verschiedene Arten organisieren:

- Ihre Ausführung kann von einer oder mehreren Bedingungen abhängig gemacht werden.
- Einzelne Anweisungen oder Anweisungsfolgen lassen sich mehrfach wiederholen.
- Anweisungsfolgen können als separate Einheiten formuliert werden, die von verschiedenen Teilen des Programms aus benutzbar sind, d.h. sie bearbeiten je nach Kontext verschiedene Daten.

Die Elemente der Programmierung

Diese Liste läßt sich unter der Verwendung von Oberbegriffen erheblich kürzer formulieren, nämlich als: *Eingabe*, *Datentypen*, *Operationen*, *Ausgabe*, *bedingte Ausführung*, *Schleifen* und *Unterprogramme*. Vollständig ist sie nicht – dafür aber mehr oder weniger allgemeingültig. Fassen wir noch einmal zusammen, diesmal unter mehr technischen Gesichtspunkten:

Eingabe: das Lesen von Daten aus einer Diskettendatei, von der Tastatur oder über eine Schnittstelle zu einem anderen Eingabegerät.

Datentypen: Konstanten, Variablen und Strukturen (»Records«), die numerische Werte (Integer- und Realzahlen), Text (einzelne Zeichen und Strings) oder Adressen (von Variablen, Routinen und Strukturen) enthalten.

Operationen: Zuweisung von Werten an Variablen, Verknüpfung von Werten (Addition, Division usw.) sowie Vergleiche (gleich, ungleich, größer als etc.).

Ausgabe: das Schreiben von Daten auf den Bildschirm, in eine Diskettendatei, zu einem Drucker oder einem anderen Ausgabegerät.

bedingte Ausführung: die Ausführung von Anweisungen nach einer Operation, wie einem Vergleich. Wenn diese Operation den Wert *True* (wahr) ergeben hat, werden die Anweisungen ausgeführt; war das Ergebnis *False* (falsch), dann werden sie übersprungen.

Schleifen: wiederholen die Ausführung von Anweisungen, wobei die Anzahl der Wiederholungen entweder fest vorgegeben ist oder von einer Bedingung abhängt (»Wiederhole, bis sich *True* / *False* ergibt«).

Unterprogramme: sind Gruppen von Anweisungen, die von mehreren Stellen des Programms *aufgerufen* werden können. Normalerweise findet dabei eine Übergabe der Daten statt, die bearbeitet werden sollen. Ein Unterprogramm kann auf dieselbe Weise wie ein vollständiges Programm aufgebaut sein.

Die nächsten sieben Abschnitte beschäftigen sich damit, wie diese grundlegenden Elemente in der Sprache Pascal realisiert sind. Wir fangen dabei mit der Ausgabe an – hauptsächlich deshalb, weil man mit einem Programm *ohne* Ausgaben *nichts* anfangen kann.

Ausgabe

Unter diesem Oberbegriff verstehen wir *jede* Art von Daten, die ein Programm von sich gibt – egal, ob es sich dabei um Text und/oder Grafiken auf dem Bildschirm/Drucker, das Schreiben in eine Datei auf der Diskette/Festplatte oder das Setzen von Schnittstellen (Ports) des Computers handelt.

Wir werden uns in diesem Abschnitt ausschließlich mit Ausgaben auf den Bildschirm beschäftigen. Detaillierte Angaben über das Schreiben von Dateien finden Sie in Kapitel 10 »Ein- und Ausgabe« des *Programmierhandbuches* und in Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuches*.

Die Prozedur Writeln

ist die wohl am häufigsten verwendete Ausgaberoutine von Pascal und schreibt Daten auf den Bildschirm. Das Format von *Writeln* ist nicht nur einfach, sondern auch sehr flexibel:

```
Writeln(Element, Element, ... );
```

Element steht dabei für »das, was ausgegeben werden soll« und kann ein numerischer Wert (*32*, *-234.118*), ein einzelnes Zeichen (*'c'*), eine Zeichenfolge (*'Hallo!'*) oder ein Wahrheitswert (*True*) sein. *Writeln* ist nicht auf direkte Angaben beschränkt – die Werte von Variablen, Konstanten, die Ergebnisse von

Funktionsaufrufen und Werte, auf die eine Zeigervariable zeigt, lassen sich genauso ausgeben, solange sie einfachen Typs (d.h. keine Datenstrukturen) sind.

Writeln gibt alle angegebenen *Elemente* der Reihe nach aus – Leerzeichen oder andere Trennzeichen werden dabei *nicht* automatisch eingefügt. Ein Aufruf dieser Prozedur wie

```
Writeln(1, 2, 3);
```

erzeugt die Ausgabe *123* auf dem Bildschirm. Um diese drei Ausgaben voneinander zu trennen, müssen Sie selbst entsprechende Leerzeichen angeben:

```
Writeln(1, ' ', 2, ' ', 3);  { ergibt 1 2 3 }
```

Einige Beispiele, in denen wir davon ausgehen, daß die Variablen *A*, *B* und *C* den Typ *Integer* haben (d.h. ganzzahlige Werte enthalten); die Variable *Name* hat den Typ *String*, speichert also eine Zeichenkette:

```
A := 1; B := 2; C := 3;
Name := 'Frank';
Writeln(A, B, C);           {  123         }
Writeln(A, ' ', B, ' ', C); {  1 2 3       }
Writeln('Hallo', Name);     {  HalloFrank  }
Writeln('Hallo, ', Name);   {  Hallo, Frank }
```

Writeln (=Write Line) erzeugt nach der Ausgabe des letzten Elements einen Zeilenvorschub, setzt also den Cursor auf den Beginn der nächsten Zeile. Die Prozedur *Write* verhält sich genauso wie *Writeln*, nur daß hier der abschließende Zeilenvorschub unterbleibt:

```
Write('Hallo, ');
Writeln(Name);      { ergibt zusammen: Hallo, Frank }
```

Über eine zusätzliche Angabe läßt sich eine *Feldbreite* für ein Element festlegen, d.h. die Anzahl von Zeichen, die *Write* und *Writeln* für dieses Element mindestens ausgeben sollen. Die Syntax ist:

```
Writeln(Element: Feldbreite, Element: Feldbreite... );
```

Feldbreite steht hier für einen Integer-*Ausdruck*: einen ganzzahligen Wert, der direkt, über eine Konstante, Variable oder als Ergebnis eines Funktionsaufrufs angegeben ist. Einige Beispiele dazu:

```
A:=1; B:= 5; C:= 100;
Writeln(A,B,C);          {  15100          }
Writeln(A:2, B:2, C:2);  {   1 5100        }
Writeln(2:B, B:B, C:B);  {     2    5  100 }
```

Writeln stellt hier also eine entsprechende Anzahl von Leerzeichen voran, um die angegebene Feldbreite zu erreichen (»rechtsbündige Formatierung«). Die Ausgabe des Wertes von *C* im zweiten Beispiel zeigt, daß *Writeln* eine angegebene Feldbreite nötigenfalls ignoriert und »überzählige« Zeichen in keinem Fall unterdrückt – die Feldbreite legt lediglich die *minimale* Breite einer Ausgabe

fest. Feldbreiten lassen sich nicht nur bei der Ausgabe ganzzahliger Werte anwenden:

```
Name := 'Frank';  B:= 10;
Writeln('Hallo', Name:8);    {  Hallo     Frank  }
Writeln('A':B, 'B');         {           AB      }
Writeln('Hallo':B, Name);    {       HalloFrank  }
```

Realzahlen (d.h. Dezimalbrüche) werden von *Write* und *Writeln* normalerweise im »wissenschaftlichen« Format und *linksbündig* ausgegeben, wobei ebenfalls eine Feldbreite verwendet werden kann.

```
Writeln(421.53);      {  4.2153000000E+02 }
Writeln(421.53:12);   {  4.21530E+02 }
Writeln(421.53:8);    {  4.2E+02 }
```

Für derartige Werte ist die Angabe eines weiteren Parameters möglich, der die Anzahl der Stellen nach dem Komma festlegt:

```
Writeln(Element:Feldbreite:Nachkommastellen, Element... );
```

Wenn die Anzahl der Dezimalstellen angegeben ist, dann verwenden *Write* und *Writeln* das Fixkomma-Format und arbeiten wie bei ganzzahligen Werten und Zeichenfolgen rechtsbündig:

```
Writeln(421.53:12:8);       {  421.53000000  }
Writeln(421.53:12:4);       {     421.5300   }
Writeln(421.53:12:0);       {          422   }
```

Wie das letzte Beispiel zeigt, wird nötigenfalls automatisch gerundet.

Datentypen

Fast alle Informationen in einem Programm gehören zu einem der fünf grundlegenden Datentypen: *Integer*, *Gleitkommazahlen*, *Text*, *Wahrheitswerte* und *Zeiger*.

Integer: Ganze Zahlen wie -1, 5, 3000 und 0.

Gleitkommazahlen: Werte, die Bruchteile ganzer Zahlen enthalten und wahlweise direkt (234.3495874) oder in wissenschaftlicher Notation ($2.345 * 1012$ – auf Computern in der Form 2.345e+12) dargestellt werden. Eine gebräuchliche Bezeichnung dieses Typs ist *Realzahlen*.

Text: Besteht aus einzelnen Zeichen (*'a'*, *'Z'*, *'!'*) und Zeichenfolgen (*'Ich bin ein String'*).

Wahrheitswerte (Booleans): können nur zwei verschiedene Werte annehmen: *True* (eine Bedingung trifft zu) und *False* (eine Bedingung trifft nicht zu).

Zeiger (Pointer): Enthalten keine direkten Werte, sondern die Startadresse eines Speicherbereichs (der seinerseits einen Wert enthalten kann) – Zeiger *zeigen* also auf Werte.

Die fünf Integer-Typen

Eine Variable des Typs *Integer* belegt zwei Bytes Speicherplatz und kann ganzzahlige Werte im Bereich von -32768 bis 32767 enthalten. Der Grund für diese Grenzen liegt darin, daß sich mit 2 Bytes 65536 unterschiedliche Werte darstellen lassen, die bei diesem Variablentyp als -32768..0..32767 interpretiert werden. Turbo Pascal definiert zusätzlich den Datentyp *Word*, der ebenfalls zwei Bytes belegt, aber vorzeichenlos interpretiert wird: mögliche Werte von *Word*-Variablen liegen folglich im Bereich von 0..65535.

Die Datentypen *ShortInt* (-128..127) und *Byte* (0..255) stellen eine Untermenge der Integer-Typen dar. Variablen dieser Typen belegen jeweils ein Byte Speicherplatz, wobei die Unterteilung von *ShortInt* in negative und positive Bereiche ebenfalls eine willkürliche Interpretation ist.

Variablen des Datentyps *LongInt* belegen 4 Bytes Speicherplatz und haben einen entsprechend großen Wertebereich – ein vorzeichenloses Gegenstück wie für *Integer* und *ShortInt* gibt es für diesen Datentyp nicht. Tabelle 2.1 faßt die Integer-Datentypen und ihre Wertebereiche zusammen:

Tabelle 2.1 Vordefinierte Integertypen und Wertebereiche

Typ	Bereich	Format
ShortInt	-128 127	1 Byte mit Vorzeichen
Integer	-32768 32767	2 Bytes mit Vorzeichen
LongInt	-2147483648 .. 2147483647	4 Bytes mit Vorzeichen
Byte	0 255	1 Byte, vorzeichenlos
Word	0 65535	2 Bytes, vorzeichenlos

Turbo Pascal erlaubt für Integer die Verwendung der hexadezimalen Notation mit einem vorangestellten Dollarzeichen ($). Ein Wert wie 255 kann also auch als $FF angegeben werden.

Die fünf Real-Typen

Bevor wir auf die technischen Hintergründe zu sprechen kommen, benutzen wir noch einmal das Beispiel NUMMER3.PAS, d.h. unser »drittes Programm« aus Kapitel 1 »Die neue integrierte Entwicklungsumgebung«:

```
program Nummer3;
var
  A, B: Integer;
  C : Integer;
begin
  Write('Geben Sie zwei Werte ein: ');
  Readln(A, B);
  C := A div B;
  Writeln('Das Ergebnis ist: ', C);
end.
```

Ist Ihnen eigentlich aufgefallen, daß dieses Programm auch bei zulässigen Eingaben alles andere als korrekt arbeitet?

Starten Sie *Nummer3* und geben Sie einmal die Werte 4 und 5 ein. Sie erhalten die Meldung

```
Das Ergebnis ist: 0
```

Der Grund: Alle drei Variablen haben den Typ *Integer* – und eine ganzzahlige Division von 4 durch 5 ergibt den Wert 0. Ändern Sie das Programm in:

```
programNummer3;  { mit einer Real-Variablen }
var
  A, B: Integer;
  C : Real;                              { 1. Veränderung }
begin
  Write('Geben Sie zwei Werte ein: ');
  Readln(A, B);
  C := A / B;                            { 2. Veränderung }
  Writeln('Das Ergebnis ist: ', C:6:2);  { 3. Veränderung }
end.
```

Nachdem die Variable *C* als *Real* deklariert und der Operator **div** durch eine (Real-)Division ersetzt worden ist, sollten Sie nun ein korrektes Ergebnis erhalten.

Eine *Realzahl* setzt sich aus drei Elementen zusammen: dem Vorzeichen, einer *Mantisse* (die den Dezimalbruch enthält) und einem Exponenten. Mit welcher Genauigkeit ein solcher Wert erfaßt werden kann, hängt davon ab, wieviele Dezimalstellen in der Mantisse gespeichert sind.

Turbo Pascal speichert Gleitkommazahlen des Typs *Real* mit 6 Bytes und stellt dabei rund 11 Dezimalstellen zur Verfügung, der zulässige Wertebereich geht von rund $+/- 10^{-38}$ bis $+/- 10^{38}$:

```
10000000000000000000     { als 1.0 E+21 gespeichert (genau) }
10000000000000990000000  { als 1.0 E+21 gespeichert (ungenau) }
239487.346           { als 2.39487346 E+05 gespeichert (genau) }
2.345938475938457    { als 2.3459384746 E+05 gespeichert (ungenau) }
283746283746.345     { als 2.8374628375 E+11 gespeichert (ungenau) }
```

Bei der Speicherung von Realzahlen ist nicht die absolute Größe für die Genauigkeit ausschlaggebend, sondern sozusagen die Anzahl und Position der verschiedenen Ziffern. Die Speicherung eines Wertes wie 1000000000000 erfordert lediglich die Zahl 1.0 in der Mantisse und einen entsprechend hohen Exponenten; ein Wert wie 2.345938475938457 hat zwar einen kleinen Exponenten, aber mehr Dezimalstellen, als in der Mantisse erfaßt werden können.

Nach der Aktivierung des Compiler-Schalters {$N+} stellt Turbo Pascal vier weitere Datentypen zur Verfügung, die größtenteils eine wesentlich höhere Genauigkeit als *Real* bieten und dem IEEE-Standard entsprechen, auf den auch der mathematische Coprozessor 8087 und seine Nachfolger zugeschnitten sind.

Wenn Ihr Computer mit einem solchen Chip ausgerüstet ist, sollten Sie grundsätzlich im Modus {$N+}arbeiten – Operationen mit Realzahlen geschehen dann wesentlich genauer *und* schneller.

Falls Sie keinen 8087-Coprozessor besitzen, müssen Sie sich trotzdem nicht unbedingt auf den Typ *Real* beschränken: Turbo Pascal enthält einen *Emulator*,

der mit dem Compiler-Befehl {$N+,E+} aktiviert wird und den Coprozessor notfalls simuliert. (Der dahinterliegende Sinn ist einfacher, als man zuerst denken mag: Programme, die im Modus {$N+,E+} compiliert worden sind, lassen sich auf Computern mit und ohne Coprozessor ausführen. Der Emulator nimmt beim Start eine Prüfung der Hardware vor: falls ein Coprozessor installiert ist, wird er direkt verwendet; falls nicht, werden die entsprechenden Befehle durch Unterprogramme ausgeführt. Das Kapitel 13 »Der Coprozessor 8087« des *Programmierhandbuches* enthält – neben einer Diskussion des Für und Wider – weitere Details).

Tabelle 2.2 faßt alle Real-Typen und ihre Wertebereiche zusammen:

Tabelle 2.2 Wertebereich und Rechengenauigkeit vordefinierter Real-Typen

Typ	Bereich	Genauigkeit	Format
Real	$2.9*10^{-39}$ bis $1.7*10^{38}$	11-12 Stellen	6 Bytes
Single	$1.5*10^{-45}$ bis $3.4*10^{38}$	7 - 8 Stellen	4 Bytes
Double	$5.0*10^{-324}$ bis $1.7*10^{308}$	15-16 Stellen	8 Bytes
Extended	$1.9*10^{-4951}$ bis $1.1*10^{4932}$	19-20 Stellen	10 Bytes
Comp	$-9.2*10^{18}$ bis $9.2*10^{18}$	18-19 Stellen	8 Bytes

Zeichen und Zeichenketten (Strings)

Einzelne Zeichen werden in Pascal mit dem Datentyp *Char* gespeichert und belegen jeweils ein Byte. *Char*-Konstanten bestehen aus einem einzelnen Zeichen, das in Apostrophen eingeschlossen ist ('a', 'x', '?', '2' usw.).

Achtung: '2' stellt das *Zeichen* »2« dar, *2* den Integerwert 2 – und *2.0* ist eine Konstante des Typs *Real*.

Eine der häufigsten Anwendungen von *Char*-Variablen dürften Rückfragen des Typs »Ja/Nein« sein, also die Ausgabe einer Frage und das Lesen eines einzelnen Zeichens von der Tastatur:

```
...
Write('Wollen Sie die Datei wirklich löschen? (J/N): ');
Readln(Ch);
if (Ch='J')or(Ch='j')  { wenn J oder j eingegeben wurde, }
   then ....           {  dann ... }
```

Hier ist *Ch* eine Variable des Typs *Char*. Was dieses Beispiel auch verdeutlichen soll: *J* und *j* sind nicht identisch – jedes Zeichen wird intern als (numerischer) ASCII-Code gespeichert (siehe Kapitel 4 »Zeichensatz und Tastaturcodes« des *Referenzhandbuches*).

Für die Angabe von Steuerzeichen, die sich innerhalb eines Quelltextes nicht direkt eingeben lassen, bietet Turbo Pascal zwei Möglichkeiten:

```
Ch := ^A;      { Ch :=   Ctrl A   }
Ch := #1;      { hat dasselbe Ergebnis }
```

Über die erste Form können alle Steuerzeichen von Ctrl A bis Ctrl Z angegeben werden (das Zeichen Esc läßt sich mit ^[darstellen). Die zweite Form arbeitet mit dem ASCII-Code des Zeichens: auf das Doppelkreuz (#) muß eine

Integer-Konstante folgen, die einen Wert im Bereich von 0..255 hat. Auf diese Weise lassen sich nicht nur Steuerzeichen, sondern auch die IBM-Grafikzeichen als *Char*-Konstanten in einen Quelltext aufnehmen.

Die Definition von Strings Ein *String* setzt sich aus einer Folge von Zeichen zusammen, die ihrerseits den Typ *Char* haben. Der Typ **string** ist in Standard-Pascal nicht definiert – in Turbo Pascal steht er zusammen mit einer Vielzahl von Routinen (Einfügen, Löschen, Vergleichen, Längenbestimmung usw.) zur Verfügung. Zur Erläuterung der technischen Hintergründe greifen wir noch einmal auf das Programm *Nummer2* aus Kapitel 1 zurück:

```
program Nummer2;
var
  Name: string[30];
begin
  Writeln('Wie heißen Sie? ');
  Readln(Name);
  Writeln('Hallo, ', Name);
end.
```

Hier wird eine Variable des Typs **string** deklariert, die maximal 30 Zeichen speichern kann. Tatsächlich reserviert der Compiler 31 Bytes für *Name* – in dem zusätzlichen Byte wird die *momentane Länge* dieses Strings festgehalten. Nur deshalb spielt es keine Rolle, wie lang der eingegebene Text ist: *Writeln* »weiß« immer, wieviele Zeichen ausgegeben werden müssen. (Wenn Sie zufällig Franz Xaver Rambrammerdenger-Osterhaas heißen sollten, müssen Sie etwas mehr Speicherplatz für *Name* reservieren – *Readln* schneidet die Eingabe in unserem Beispiel nach dem 30. Zeichen ab).

Da zur Speicherung der momentanen Länge ein einzelnes Byte verwendet wird, kann ein String maximal Platz für 255 Zeichen bieten. Die Deklaration ohne eine Längenangabe nach **string** erzeugt in Turbo Pascal eine solche String-Variable, hat also dieselbe Wirkung wie `string[255]`

String-Konstanten bestehen aus einer Zeichenfolge, die in Apostrophen eingeschlossen werden muß. Ein Apostroph *innerhalb* einer Zeichenfolge wird durch zwei aufeinanderfolgende Apostrophen dargestellt:

```
var
  s : string;
begin
  s := 'Hallo, Freunde!';
  Writeln(s);              {  Hallo, Freunde!  }
  s := 'That''s it!';
  Writeln(s);              {  That's it!       }
```

Über die Angabe von *Char*-Konstanten mit ^ und # lassen sich auch ein oder mehrere Steuerzeichen in eine String-Konstante aufnehmen:

```
  s := 'Aufwachen!'^G^G^G;
  Writeln(s);                    { Aufwachen! <Piep> <Piep> <Piep> }
  s := 'Hey! '^G^G'Aufwachen!';
```

```
Writeln(s);                    { Hey! <Piep> <Piep> Aufwachen!   }
s := 'Hey! '#7#13#10'Aufwachen!';
Writeln(s);                    { Hey! <Piep>
                                 Aufwachen!   }
```

Die Steuerzeichen müssen außerhalb der Apostophen stehen und lückenlos aufeinander folgen (Leerzeichen dazwischen sind nicht erlaubt). Eine Besprechung der Routinen, die Turbo Pascal für die Manipulation von Strings zur Verfügung stellt, finden Sie in Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuches*.

Der Datentyp Boolean

Variablen dieses Typs belegen ein Byte im Speicher und können nur zwei Werte annehmen: *True* (eine Bedingung trifft zu) und *False* (trifft nicht zu). Vergleiche und andere Prüfungen wie

```
if X = 4          { wenn X den Wert 4 hat ...}
  then....        { dann führe diese Anweisung aus }

Done := False;    { wir sind noch nicht fertig }
repeat            { wiederhole ....}
  ...               { diese Anweisungen }
untilDone;        { bis Done den Wert True hat }
```

erzeugen boolesche Werte als Ergebnis. Die Bezeichner *True* und *False* sind vordefinierte Konstanten und können booleschen Variablen direkt zugewiesen werden.

Additionen oder andere arithmetische Operationen sind mit booleschen Werten nicht erlaubt – dafür lassen sich aber die Ergebnisse mehrerer Vergleiche miteinander verknüpfen:

```
if (X = 4) and (Z = 5)   { wenn X den Wert 4 hat und Z den Wert 5
...}
```

Wir kommen auf Verwendung dieses Datentyps im Zusammenhang mit bedingter Ausführung und Schleifen noch ausführlich zu sprechen.

Zeiger-Typen

Ein Zeiger enthält im Gegensatz zu den restlichen Datentypen von Pascal keinen direkten Wert, sondern die *Adresse* eines Wertes – er zeigt also auf einen Speicherbereich, in dem ein Wert gespeichert ist. Fangen wir auch hier mit einem Beispiel an:

```
var
  Puffer: string;
  Pufferzeiger: ^string;
```

Im Quelltext unterscheiden sich diese beiden Deklarationen nur durch ein einziges Zeichen (^) – die technischen Unterschiede zwischen *Puffer* und *Pufferzeiger* sind dagegen gewaltig. *Puffer* ist eine Variable vom Typ **string**, für die der Compiler 256 Byte Platz reserviert hat, *Pufferzeiger* dagegen eine Zeigervariable, die 4 Bytes Speicherplatz belegt. Diese Variable *kann* auf einen String mit 255

Zeichen zeigen – aber erst, nachdem wir einen solchen String erzeugt haben. Dazu wird die Prozedur *New* verwendet:

```
New(Pufferzeiger);
```

Dieser Aufruf belegt einen 256 Byte großen Speicherbereich und weist *Pufferzeiger* die Startadresse dieses Bereichs zu. Zugriffe auf den soeben belegten Bereich geschehen über die Zeigervariable, nämlich als

```
Pufferzeiger^ := 'Dieser String wird in dem Bereich gespeichert,' +
                 ' auf den Pufferzeiger zeigt';
Writeln(Pufferzeiger^);
Dispose(Pufferzeiger);  { gibt den belegten Bereich wieder frei }
```

Wir kommen in Kapitel 4 »Programmierung für Fortgeschrittene« noch ausführlich auf die Möglichkeiten von Zeigern zurück. Eine Erläuterung der technischen Hintergründe finden Sie in Kapitel 14 »Interne Details« des *Programmierhandbuches*.

Bezeichner

Bisher haben wir unbekümmert Variablennamen vergeben, ohne dabei auf eventuelle Beschränkungen zu achten. Bevor wir uns an eine formale Definition wagen, definieren wir erst einmal den Begriff *Bezeichner* selbst anhand der bis jetzt verwendeten Beispiele:

begin, **end**	reservierte Wörter von Pascal
Name, A, B	Variablennamen
Nummer1	Programmname
Write, Readln	vordefinierte Routinen von Turbo Pascal

Turbo Pascal hat – wie jede Sprache – einige Regeln, die bei der Vergabe von Bezeichnern beachtet werden müssen:

- Bezeichner müssen mit einem Buchstaben (A..Z, a..z) oder einem Unterstrich (_) beginnen.
- Auf dieses erste Zeichen können weitere Buchstaben, Ziffern (0..9) und Unterstriche folgen. Die deutschen Umlaute sind nicht zulässig, genausowenig wie das »ß«.
- Die ersten 63 Zeichen eines Bezeichners sind signifikant. Das bedeutet: Wenn die ersten 63 Zeichen zweier Bezeichner identisch sind, werden diese Bezeichner als identisch behandelt – unabhängig davon, ob auf diese 63 Zeichen noch weitere (eventuell unterschiedliche) Zeichen folgen.
- Zwischen Groß- und Kleinschreibung wird nicht unterschieden – Angaben wie *Index*, *INDEX* und *iNdEx* sind für den Compiler identisch.
- Reservierte Wörter (wie z.B. **for** oder **while**) können nicht als Bezeichner verwendet werden.

Operatoren

Der Hauptzweck eines Programms besteht nicht in der Speicherung von Daten, sondern in ihrer Verknüpfung und Manipulation. Deshalb bietet Turbo Pascal gleich acht verschiedene Operatorentypen, mit denen sich derartige Operationen ausführen lassen. Die meisten dieser Operatoren sind *binär*, d.h. sie verknüpfen zwei Operanden (wie in $a + b$) miteinander. Die Regeln sind dieselben wie in der Schulmathematik: Multiplikation kommt vor Addition, durch Einklammern läßt sich die Reihenfolge der Rechenschritte beliebig verändern. Im Zusammenhang mit der *Rangfolge* von Operatoren (siehe Tabelle 2.3) gelten drei grundlegende Gesetze:

- Ein Operand zwischen zwei Operatoren von unterschiedlichem Rang ist immer an den höherrangigen Operator gebunden.
- Ein Operand zwischen zwei gleichrangigen Operatoren ist immer an den Operator gebunden, der links von ihm steht.
- Ausdrücke in Klammern werden als ein einzelner Operand betrachtet und immer als erstes ausgewertet.

Tabelle 2.3 Rangfolge der Operatoren

Operatoren	Rangfolge	Kategorie
not	1 (am höchsten)	unär
*, /, **div**, **mod**, **shl**, **shr**	2	multiplizierend
+, -, **or**, **xor**	3	addierend
=, <>, <, >, <=, >=, **in**	4	relational

Operationen von gleichem Rang werden normalerweise von links nach rechts durchgeführt, der Compiler kann allerdings zur Optimierung des Rechenweges Umstellungen vornehmen.

Einige Beispiele dazu:

```
3 + 4 * 7       { a) 4 * 7;   b) 3 + 28;   Ergebnis: 31  }
3 + (4 * 7)     { dasselbe Ergebnis }
(3 + 4) * 7     { a) 3 + 4;   b) 7 * 7;    Ergebnis: 49  }
8 div 4 + 2     { a) 8 div 4; b) 2 + 2;    Ergebnis:  4  }
8 div 4 * 2     { dasselbe, weil div links von * steht   }
8 div (4 * 2)   { a) 4 * 2;   b) 8 div 8;  Ergebnis:  1  }
```

Zuweisungen

stellen die grundlegende Operation jeder Programmiersprache dar. Pascal verwendet einen Doppelpunkt, gefolgt von einem Gleichheitszeichen (:=), um einer *Variablen* auf der linken Seite den *Wert* der rechten Seite zuzuweisen. Zum Beispiel:

```
Summe := A + B;
Division: = A / B;
```

Diese beiden Anweisungen berechnen jeweils zuerst den Ausdruck rechts des Gleichheitszeichens (*A + B* bzw. *A / B*) und weisen dann das *Ergebnis* der Variablen zu, deren Bezeichner links des Gleichheitszeichens angegeben ist (also *Summe* bzw. *Division*).

Unäre und binäre Operatoren

Neben der üblichen Menge binärer Operatoren, die auf Integer- und Realwerte anwendbar sind, also

*	Multiplikation
div	Integer-Division
/	Real-Division
mod	Modulo
+	Addition
-	und Subtraktion

unterstützt Turbo Pascal ein *unäres Minus* (a + (-b)) zur Bildung des *Zweierkomplements* eines Integer-Operanden; und der Vollständigkeit halber ein *unäres Plus* (a + (+b)), das keine weitere Wirkung hat.

Bitmanipulationen

können in Turbo Pascal über sechs verschiedene Operatoren ausgeführt werden:

shl	»shift left« – Linksschieben um eine bestimmte Anzahl von Bitpositionen, die freiwerdenden Bits werden auf »0« gesetzt.
shr	»shift right« – Rechtsschieben um eine bestimmte Anzahl von Bitpositionen und Setzen der freiwerdenden Bits auf »0«.
and	bitweises UND
or	bitweises ODER
xor	bitweise Antivalenz
not	bitweise Negation

Einige Beispiele, bei denen Sie versuchen sollten, das Ergebnis zuerst selbst zu bestimmen:

```
x := 8 shl  2;       { 32 }
x := $1FF shl 7;     { $FF80 (= dezimal 65408) ä nur für Word }
x := 9 shr  1;       { 8, das niederwertigste Bit geht verloren }
x := $FF00 shr 8;    { $00FF }
x := 8 and  2;       { 0 }
x := $FF and 8;      { $08 }
x := 8 or 2;        { $0A (= dezimal 10) }
x := $FF or 8;       { $FF }
x := 8 xor  2;       { $0A (= dezimal 10) }
x := $FF xor 8;      { $F7 (= dezimal 247) }
y := not  8;         { $FFF7 ( = dezimal -9 (!)) - nur Integer }
y := not  $FF;       { $FF00 ( = dezimal -256) - nur Integer }
```

Wenn *x* als *Integer* deklariert ist, meldet der Compiler für die zweite Zuordnung einen Bereichsüberlauf (der Wert würde als *Integer* eine negative Zahl

ergeben). Analoges gilt für eine Deklaration von *y* als *Word* und den Operator **not**: **not** verkehrt das Vorzeichen und ist nicht auf *Word*-Variablen anwendbar, weil dieser Typ keine negativen Werte definiert. (Die sich ergebenden Bitmuster wären allerdings für beide Typen dieselben. Wie im Abschnitt über die Integer-Typen erläutert, ist die Unterteilung eines Wortes mit 16 Bit in positive und negative Wertebereiche eine Interpretation).

Relationale Operatoren

vergleichen ihre Operanden miteinander, liefern ein boolesches Ergebnis zurück und lassen sich in Turbo Pascal auf alle Grundtypen (numerische Werte, Char, Strings und Zeiger) anwenden. Die folgenden Operatoren sind definiert:

`>`	größer als
`>=`	größer oder gleich
`<`	kleiner als
`<=`	kleiner oder gleich
`=`	gleich
`<>`	ungleich
in	Prüfung einer Mengenzugehörigkeit

Der Operator **in** prüft, ob ein angegebenes Element in einer Menge vorhanden ist und liefert ebenfalls ein boolesches Ergebnis – Details dazu finden Sie in Kapitel 6 »Ausdrücke« des *Programmierhandbuches*. Ein Beispielprogramm zu den anderen Operatoren:

```
program Relational;
var
  A, B: Integer;
begin
  Write('Geben Sie zwei Werte ein: ');
  Readln(A, B);
  if A = B then Writeln('Beide Werte sind gleich');
  if A <> B
  then begin
     Write('Die beiden Werte sind ungleich: ');
     if A < B then Writeln(A, ' ist kleiner als ', B);
     if A > B then Writeln(A, ' ist größer als ', B);
   end;
end.
```

Das Ergebnis eines Vergleichs läßt sich auch direkt einer booleschen Variablen zuweisen. Nach einer entsprechenden Deklaration wie

```
var
  IstKleiner : Boolean;
```

können wir schreiben:

```
...
IstKleiner := (A < B);
```

```
if IstKleiner then Writeln(A, ' ist kleiner als ', B);
...
```

Logische Operatoren

haben dieselben Namen wie die Operatoren zur Bitmanipulation: **and**, **or**, **xor** und **not**. Sie arbeiten allerdings nicht bitweise, sondern logisch, d.h. verknüpfen boolesche Werte miteinander. Modifizieren wir das Beispielprogramm aus dem vorangegangenen Abschnitt:

```
var
  A, B       : Integer;
  IstKleiner : Boolean;
begin
  Write('Geben Sie zwei Werte ein: ');
  Readln(A, B);
  if (A = B) and (A = 0) { Wenn A = B ist UND den Wert 0 hat ... }
    then Writeln('Beide Werte sind 0');
  if (A = 0) or (B = 0)  { Wenn A = 0 ist ODER B = 0 ist ... }
    then Writeln('Mindestens einer der Werte ist 0');

  IstKleiner := (A < B);
  if not IstKleiner   { wenn IstKleiner nicht True ist ...}
    then Writeln(A, ' ist nicht kleiner als ', B);
  ...
```

Wir haben die Vergleiche hier in Klammern gesetzt, weil **and** eine höhere Rangstufe als = hat (vgl. Tabelle 4.3). Bei einer Formulierung ohne Klammern, also

```
if A = B andA = 0 ...
```

würde der Compiler ansonsten als erstes den Ausdruck *B* **and** *A* bewerten, danach mit *A* vergleichen und schließlich mit einem Syntaxfehler abbrechen. (Warum? Weil er mit dem folgenden *=0* nichts mehr anfangen kann).

Zwischen logischen und bitweisen Operatoren gibt es also einige Unterschiede:

- Logische Operatoren erzeugen boolesche Ergebnisse, bitweise Operatoren erzeugen numerische Ergebnisse.

- Logische Operatoren lassen sich nicht direkt mit bitweisen Operatoren kombinieren. Wenn *Flag* eine boolesche Variable ist und *Index* den Typ *Integer* hat, dann weist der Compiler einen Ausdruck wie *Flag* **and** *Index* als illegal zurück. Eine Formulierung wie *Flag* **and** (*Index* **or** *$10* =$30) ist dagegen legal.

- Die logischen Operatoren **and** und **or** arbeiten in Turbo Pascal standardmäßig mit einem *Kurzschlußverfahren* – zusammengesetzte Bedingungen werden nur soweit bewertet, bis das Gesamtergebnis feststeht. Details dazu finden Sie in Kapitel 2 »Compiler-Befehle« des *Referenzhandbuches* und in Kapitel 6 »Ausdrücke« des *Programmierhandbuches*.

Adreß-Operatoren
Turbo Pascal definiert zwei spezielle Operatoren für die Arbeit mit Zeigern und Adressen:

@ »Adresse von« – liefert die Adresse des angegebenen Bezeichners zurück. Wenn *Summe* eine Variable beliebigen Typs ist, dann liefert *@Summe* die Adresse zurück, ab der der Inhalt von *Summe* gespeichert ist.

^ »Indirektion« – liefert den Inhalt der Speicherzelle(n) zurück, auf den ein Zeiger zeigt. Wenn *Pufferzeiger* ein Zeiger auf einen String ist, dann liefert *Pufferzeiger^* den Inhalt dieses Strings.

Mengen-Operatoren
sind ebenfalls ausführlich in Kapitel 6 »Ausdrücke« des *Programmierhandbuches* beschrieben und deshalb hier nur als Tabelle wiedergegeben:

+	Vereinigung zweier Mengen
-	Differenz zweier Mengen
*	Querschnitt zweier Mengen

String-Operatoren
Das Pluszeichen verbindet zwei Stringteile miteinander und ist der einzige in Turbo Pascal definierte String-Operator. Die folgenden Zuweisungen sind identisch:

```
s := 'Teil1' + 'Teil2';
s := 'Teil1Teil2';
```

Eingabe

Turbo Pascal enthält mehrere Routinen, mit denen Daten von der Tastatur, aus einer Diskettendatei oder über Schnittstellen des Computers gelesen werden können. Wir beschäftigen uns in diesem Abschnitt ausschließlich mit *Read* und *Readln* zum Lesen der Tastatur – detaillierte Informationen über das Lesen von Dateien und Zugriffe auf Peripheriegeräte finden Sie in Kapitel 10 »Ein- und Ausgabe« des *Programmierhandbuches* und in Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuches*.

Die Prozedur Readln
hat eine ähnliche Syntax wie *Writeln*:

```
Readln(Element, Element... );
```

Element steht hier für eine einfache Variable beliebigen Typs außer *Boolean* (also *Char*, *String*, einen Integer- oder Real-Typ) und bekommt von *Readln* den eingegebenen Wert zugeordnet. *Readln* erwartet einen Zeilenvorschub nach der Eingabe aller *Elemente*. Wenn *s* eine Variable des Typs **string** ist, dann liest also ein Aufruf wie

```
Readln(s);
```

so lange Zeichen von der Tastatur, bis ein Enter eingegeben wird. »Überzählige« Zeichen werden automatisch ignoriert: Wenn *Ch* eine Variable des Typs *Char* ist, dann liest der Aufruf

```
Readln(Ch);
```

zwar ebenfalls Zeichen von der Tastatur bis zum nächsten Enter, ordnet *Ch* aber nur das erste eingegebene Zeichen zu. Wie bereits erwähnt, arbeitet diese automatische Begrenzung auch für Strings: Wenn *s* als **string**[30] deklariert ist, dann werden *s* bei einem Aufruf von *Readln* maximal die ersten 30 Zeichen der Eingabe zugeordnet.

Wenn mehr als ein Element vom Typ *Char* angegeben ist, liest *Readln* die entsprechenden Zeichen direkt hintereinander. Ein Aufruf wie

```
Readln(Ch_1, Ch_2);
```

ordnet bei einer Eingabe wie

ABC Enter

der Variablen *Ch_1* das Zeichen *A* und der Variablen *Ch_2* das Zeichen *B* zu. Weitere Zeichen bis zum abschließenden Enter (wie in unserem Beispiel das *C*) werden auch hier als »überzählig« behandelt, d.h. ignoriert.

Wenn mehr als ein numerisches Element angegeben ist, erwartet *Readln* entsprechende Trennzeichen (Tabs, Leerzeichen, Enter) in der Eingabe. Der Aufruf *Readln*(*A, B*), den wir in den vorangegangenen Beispielen ständig verwendet haben, erwartet also eine der folgenden Eingaben:

```
44      25 Enter    <-- Leerzeichen als Trennung
44 ↓|   25 Enter    <-- Tabulator als Trennung
44 Enter 25 Enter   <-- Enter  als Trennung
```

Das Lesen mehrerer Strings mit einem Aufruf von *Readln* ist **nicht** möglich. Bei einem Aufruf wie

```
Readln(String_Var1, String_Var2);
```

ordnet *Readln* sämtliche Zeichen bis zum abschließenden Enter der Variablen *String_Var1* zu, *String_Var2* bekommt einen Nullstring (d.h. eine leere Zeichenkette) zugewiesen.

Weitere Details zu Readln (und zur Prozedur *Read*) finden Sie in den entsprechenden Abschnitten in Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuches*.

Bedingte Ausführung

Die Möglichkeit, bestimmte Operationen abhängig von Bedingungen auszuführen, ist die Grundlage »entscheidungsfähiger« Programme – ohne sie wäre der Computer bestenfalls in der Lage, eine vorgegebene Folge von Anweisungen zu reproduzieren. Wir haben die einfachste Form bedingter Anweisungen in den

vorangegangenen Beispielen bereits des öfteren verwendet, ohne groß auf Formales einzugehen, und holen das nun nach:

Die Anweisung if

Die grundlegende Struktur von **if** sieht so aus:

```
if (Ausdruck)
   then Anweisung_1
   else Anweisung_2;
```

Ausdruck steht hier für einen beliebigen Ausdruck, dessen Ergebnis ein boolescher Wert sein muß. Wenn sich *Ausdruck* als *True* ergibt, wird *Anweisung_1* ausgeführt (und *Anweisung_2* übersprungen); ansonsten wird *Anweisung_2* ausgeführt (und *Anweisung_1* übersprungen).

Der **else**-Zweig einer **if**-Anweisung ist optional, d.h. er muß nicht angegeben werden. Bei einer Anweisung wie

```
if (Ausdruck)
  then  Anweisung_1;
```

wird *Anweisung_1* ebenfalls nur dann ausgeführt, wenn sich *Ausdruck* als *True* ergibt. Ist *Ausdruck* dagegen *False*, dann geht die Programmausführung einfach mit dem nächsten Schritt hinter *Anweisung_1* weiter.

Hinweis: *Anweisung_1* wird nur dann mit einem Semikolon abgeschlossen, wenn kein **else**-Zweig folgt. Auf diese Weise werden Mehrdeutigkeiten bei verschachtelten **if**-Anweisungen vermieden, die in Kapitel 7 »Anweisungen« des *Programmierhandbuches* erläutert sind.

Anweisungsblöcke

Die Syntax von **if** läßt jeweils eine einzige Anweisung für jeden Zweig zu. Ein Konstrukt wie

```
if (Ausdruck)
 then Anweisung_1;
      Anweisung_2     { FALSCH ! }
  else Anweisung_3;
```

erzeugt einen Fehler bei der Compilierung. Die Lösung bei derartigen Problemstellungen liegt in der Verwendung von *Anweisungsblöcken* (compound statements). Ein Anweisungsblock besteht aus

- dem reservierten Wort **begin**,
- gefolgt von einer beliebigen Anzahl von Anweisungen, die jeweils mit einem Semikolon abgeschlossen werden,
- und einem abschließenden **end**.

Anweisungsblöcke lassen sich beliebig verschachteln und können überall dort stehen, wo auch eine einzelne Anweisung zulässig wäre (mit einer Ausnahme: innerhalb von Ausdrücken sind sie nicht erlaubt).

An diesem Punkt sollte auch klar werden, worum es sich bei einem kompletten Pascal-Programm handelt: um einen Anweisungsblock.

Die case-Anweisung

Das folgende Listing stellt einen ersten Versuch dar, die oberste Menüebene der integrierten Entwicklungsumgebung nachzuempfinden:

```
var
  Ch : Char;     { speichert einen Tastendruck }
begin
  Readln(Ch);    { Eingabe des Befehls }
  if (Ch = 'F') or (Ch = 'f') then DoFileMenu
    else if (Ch = 'E') or (Ch = 'e') then StartEditor
    else if (Ch = 'R') or (Ch = 'r') then DoRun
    ....
```

Selbstverständlich funktioniert dieses Konstrukt – aber sehen Sie sich einmal die folgende Formulierung an:

```
var  Ch : Char;
begin
  Readln(Ch);
  case Ch of
    'F', 'f': DoFileMenu;     { Menü File aktivieren }
    'E', 'e': StartEditor;    { Editor aktivieren }
    'R', 'r': DoRun;          { "Run" ausführen }
    'C', 'c': DoCompileMenu;  { Menü Compile aktivieren }
    'O', 'o': DoOptionsMenu;  { Menü Options aktivieren }
  end;
  ...
```

Eine **case**-Anweisung besteht aus einem zu bewertenden Ausdruck (in unserem Beispiel die Variable *Ch*) und einer beliebig langen Folge von **case**-Marken, auf die jeweils eine einzelne Anweisung oder ein Anweisungsblock folgt. Bei der Ausführung des Programms wird der Wert des Ausdrucks der Reihe nach mit den angegebenen Marken verglichen. Wenn beide übereinstimmen, dann folgen zwei Schritte:

- die auf die Marke folgende Anweisung wird ausgeführt,
- danach wird der Rest der **case**-Anweisung übersprungen.

Der zu bewertende Ausdruck wird *Selektor* genannt und kann einen beliebigen ordinalen Typ (Integer, *Char*, logisch) haben, solange dieser Typ mit dem der angegebenen **case**-Marken übereinstimmt. Anstelle einzelner Marken sind auch mehrere Marken oder Bereichsangaben zulässig:

```
case Ch of
  '0'..'9'   : Writeln('Sie haben eine Ziffer eingegeben');
  '?','!'    : Writeln('Sie haben ein "!" oder "?" eingegeben');
  'A'..'Z',
  'a'..'z'   : begin   { ein Anweisungsblock }
                 Write('Ein Buchstabe: ');
                 Writeln (Ch, Ch, Ch);
               end;   { Ende des Anweisungsblocks }
```

```
    else
        Writeln('Das war irgendetwas anderes.');
        Writeln ('Vielleicht ein Umlaut?');
end;  { der case-Anweisung }
```

Was dieses Beispiel noch zeigen soll: Bei einer **case**-Anweisung ist in Turbo Pascal ein (optionaler) **else**-Zweig möglich, dessen Anweisungen nur ausgeführt werden, wenn alle vorherigen Vergleiche fehlschlagen. Dieser **else**-Zweig wird von vornherein als Anweisungsblock betrachtet – hier können also mehrere Anweisungen ohne einleitendes **begin** stehen. Noch einige technische Details:

- Alle **case**-Marken müssen Konstanten oder Bereiche von Konstanten sein und denselben Typ wie der Selektor haben. Prüfungen unterschiedlichen Typs oder Bedingungen wie

```
casex of  { x ist eine numerische Variable }
  '0'..'9' : ...        Zeichen!
   < 4     : ...        relationaler Operator!
  ...
```

werden vom Compiler als Typ-Verletzung bzw. Syntaxfehler zurückgewiesen. Richtig formuliert müßte dieses Beispiel lauten:

```
casex of  { x ist eine Char-Variable }
'0'..'9' : if (x='4') then ...
  ...
```

- Jede Marke darf nur einmal innerhalb einer **case**-Anweisung erscheinen. Eine Formulierung wie

```
caseCh of
  '0'..'9' :  ...
  '4'      :  ....
  ...
```

wird vom Compiler als fehlerhaft ausgewiesen (und wäre sinnlos – auch wenn *Ch* hier das Zeichen *'4'* enthält, wird der als erstes angegebene Zweig ausgeführt und der Rest der **case**-Anweisung übersprungen).

- Realzahlen und Strings, Arrays und alle anderen *Datenstrukturen* sind als Selektor nicht zulässig.

Schleifen

führen eine Anweisung bzw. einen Anweisungsblock wiederholt aus. Pascal stellt drei mögliche Arten von Schleifen zur Verfügung: **while**, **repeat..until** und **for**. Wir werden sie in dieser Reihenfolge besprechen.

Schleifen mit while

prüfen einen logischen Ausdruck *vor* der Ausführung der folgenden Anweisungen. Wenn der Ausdruck *True* ergibt, werden die Anweisungen ausgeführt;

ergibt er *False*, dann wird das Programm mit der nächsten Anweisung hinter der Schleife fortgesetzt. Ein Beispiel:

```
program whileDemo1;
var
  x : Integer;
begin
  x := 0;
  while (x < 10) do    { solange x kleiner als 10 ist ...}
  begin
    Write(x:4);        { wird dieser Anweisungsblock wiederholt }
    x := x + 1;
  end;
  Writeln('  Ende');
end.
```

Die Variable *x* hat vor dem ersten Durchlauf der Schleife den Wert 0 und der Vergleich *x* <10 ergibt *True* – folglich wird der Anweisungsblock ausgeführt. Da der Wert von *x* innerhalb des Anweisungsblocks jeweils um eins erhöht wird, ergibt der Vergleich nach dem zehnten Durchlauf *False*. Die Ausgaben dieses Programms sind:

```
 0   1   2   3   4   5   6   7   8   9 Ende;
```

Was passiert, wenn wir *x* zu Anfang des Programms auf den Wert 10 setzen? Sie erhalten die Ausgabe

```
Ende
```

und sonst nichts – weil der Vergleich bereits vor dem ersten Durchlauf der Schleife *False* ergibt, wird der Anweisungsblock kein einziges Mal ausgeführt. Was passiert, wenn wir *x* innerhalb der Schleife überhaupt nicht erhöhen? Sie erhalten die Ausgabe

```
0   0   0   0   0   0   0   0   0   0   0   0   0   0 .....
```

Die formale Definition von **while** ist:

```
while(Ausdruck) do
  Anweisung;
```

Wie bereits demonstriert, steht Anweisung hier (analog zu **if**) entweder für eine einzelne Anweisung oder einen Anweisungsblock und wird auch als *Rumpf* der Schleife bezeichnet.

Schleifen mit repeat...until

Dieses Konstrukt unterscheidet sich von einer **while**-Schleife in drei Punkten:

- Die Prüfung findet nicht zu Beginn, sondern am Ende des Schleifenrumpfes statt.
- Der Schleifenrumpf wird nicht wiederholt, *solange* der Ausdruck *True* ergibt, sondern *bis* zu diesem Punkt.

- Der Schleifenrumpf wird von vornherein als Anweisungsblock betrachtet, ein entsprechendes **begin** und **end** ist nicht notwendig.

Modifizieren wir das zuvor verwendete Programm ein wenig:

```
program repeatDemo;
var
  x : Integer;
begin
  x := 0;
  repeat              { wiederhole }
    Write(x:4);       { diese beiden }
    x := x + 1;       { Anweisungen }
  until (x >= 10);    { bis x den Wert 10 erreicht hat }
  Writeln('  Ende');
end.
```

Die Ausgaben dieses Programms sind dieselben wie bei **while** gezeigt. Da die Prüfung mit »umgekehrtem Vorzeichen« stattfindet, lautet die entsprechende Bedingung hier »bis *x* einen Wert größer als 9 hat« (und nicht »solange *x* kleiner als 10 ist«).

Da die Prüfung aber am Ende des Schleifenrumpfes stattfindet, wird der Rumpf auf jeden Fall einmal wiederholt. Setzen Sie *x* einmal zu Anfang des Programms auf den Wert 10000: Sie erhalten die Ausgabe

```
10000  Ende
```

Ein weiteres Beispiel:

```
program repeatDemo2;
var
  Ch: Char;
begin
  repeat                 { wiederhole }
    Read(Ch);            { ein einzelnes Zeichen lesen }
    Write(Ch);           { und ausgeben }
  until (Ch = 'x');      { bis ein 'x' gelesen/ausgegeben wurde }
end.
```

Dieses Programm liest also solange Zeichen von der Tastatur und gibt sie aus, bis ein 'x' eingegeben wurde. Es ließe sich auch mit **while** formulieren – allerdings etwas mühsamer:

```
program whileDemo2;
var
  Ch: Char;

begin
   Read(Ch);           { wir müssen hier zuerst ein Zeichen lesen,
                         damit es anschließend geprüft werden kann }
   while (Ch <> 'x') do  { solange das Zeichen kein 'x' ist }
   begin
```

```
    Write(Ch);      { Zeichen ausgeben }
    Read(Ch);       { nächstes Zeichen einlesen }
  end;
  Write(Ch);        { das Zeichen war ein 'x' und
                      wird als letztes ausgegeben }
end.
```

Die formale Definition von **repeat..until**:

```
repeat
  Anweisung;
  Anweisung;
  ....
until (Ausdruck);
```

Schleifen mit for

sind dann die beste Wahl, wenn die Anzahl der Wiederholungen von vornherein feststeht. Unser Demonstrationsprogramm *whileDemo1* ließe sich also recht einfach mit einer **for**-Schleife formulieren:

```
program forDemo;
var
  x : Integer;
begin
  for x := 0 to 9 do
    Write(x:4);
  Writeln('  Ende');
end.
```

Die Variable *x* wird hier als *Laufvariable* der Schleife bezeichnet. Bevor wir auf die Einzelheiten zu sprechen kommen, erst einmal die formale Definition:

```
for Laufvariable := Startwert to Endwert do
  Anweisung;
```

for führt eine ganze Reihe von Schritten automatisch aus:

1. Der *Laufvariablen* wird der als *Startwert* angegebene Wert zugewiesen.

2. Darauf folgt ein Vergleich mit dem *Endwert*: wenn die Laufvariable kleiner oder gleich diesem Wert ist, wird *Anweisung* ausgeführt.

3. Nach der Ausführung der *Anweisung* erhöht **for** die Laufvariable um eins und wiederholt den Schritt 2 – solange, bis die Laufvariable einen größeren Wert als *Endwert* hat.

Anweisung kann wie üblich entweder eine einzelne Anweisung oder ein mit **begin** und **end** eingeschlossener Anweisungsblock sein. Der Typ der *Laufvariablen* ist nicht auf *Integer* beschränkt – *Char* oder ein beliebiger Aufzählungstyp sind genauso möglich. *Startwert* und *Endwert* müssen Ausdrücke mit einem Typ sein, der zu dem der Laufvariablen kompatibel ist. Ein Beispiel, in dem eine Laufvariable des Typs *Char* verwendet wird:

```
var
  Ch: Char;
begin
  for Ch := 'a' to 'z' do
   Write(Ch);            { abcdefghijklmnopqrstuvwxyz }
end.
```

Was passiert, wenn der Startwert (und damit der Wert der Laufvariablen) bereits vor dem ersten Durchlauf des Schleifenrumpfes größer als der Endwert ist? Dasselbe wie bei **while**: Der Rumpf wird überhaupt nicht ausgeführt.

Über den Zusatz **downto** anstelle von **to** läßt sich festlegen, daß die Laufvariable nach jedem Durchlauf nicht erhöht, sondern erniedrigt wird. Dementsprechend findet die Prüfung gegen den *Endwert* mit umgekehrtem Vorzeichen statt:

```
var
  Ch: Char;
begin
  for Ch := 'z' downto 'a' do
   Write(Ch);            { zyxwvutsrqponmlkjihgfedcba }
end.
```

Prozeduren und Funktionen

Versuchen wir uns einmal an einem Programmstück, das Kleinbuchstaben in Großbuchstaben umwandelt und dabei auch die deutschen Umlaute berücksichtigt:

```
if (Ch >= 'a') and (Ch <= 'z')
   then Ch := chr(ord(ch) - 32)    { 'a' -> 'A', 'b'--> 'B'... }
   else if Ch = 'ä' then Ch := 'Ä'
   else if Ch = 'ö' then Ch := 'Ö'
   else if Ch = 'ü' then Ch := 'Ü'
   else ;   { kein Kleinbuchstabe --> keine Änderung }
```

Wenn diese Umwandlung an mehreren Stellen des Programms benötigt wird, kann man sie natürlich einfach wiederholen – schließlich hat der Editor von Turbo Pascal eine Funktion, mit der man Textblöcke vervielfachen kann...

Unterprogramme lösen das Problem, ein und dieselbe Folge von Anweisungen an *verschiedenen Stellen* eines Programms ausführen zu lassen. Wir haben beispielsweise *Writeln* bereits recht oft verwendet, ohne dabei zu erklären, daß der Compiler nicht jedesmal den Code dieser Prozedur wiederholt, sondern nur einen *Aufruf* einsetzt, wobei auszugebende Werte und Variablen als *Parameter* übergeben werden.

Einige formale Definitionen:

- Eine *Prozedur* bekommt bei ihrem Aufruf Werte und Variablen als Parameter übergeben und führt eine Reihe von Anweisungen aus, die als *Block* dieser

Prozedur definiert sind. Prozeduren können übergebene Variablen verändern (wie *Readln*) oder nur verwerten (wie *Writeln*).

- Eine *Funktion* unterscheidet sich von einer Prozedur dadurch, daß sie zusätzlich ein *Funktionsergebnis* zurückliefert.

Bevor wir uns weiter darum kümmern, wie sich eine Folge von Anweisungen als Prozedur oder Funktion formulieren läßt, müssen wir allerdings erst einmal kurz auf die Gesamtstruktur eines Pascal-Programms eingehen.

Die Struktur eines Programms
sieht im Standard-Pascal folgendermaßen aus:

```
program <Programmname>;
label
  Labelname, Labelname...;
const
  Konstanten-Bezeichner = Wert; ...
type
  Typ-Bezeichner = <Typ-Definition>;  ...
var
  Variablen-Bezeichner : Typ-Bezeichner;  ...

  < Prozeduren und Funktionen >

begin
  < Hauptprogramm >
end.
```

Die ersten fünf Teile (von **label** bis *<Prozeduren und Funktionen*) müssen nicht alle in einem Programm erscheinen – wenn sie es aber tun, dann ist die gezeigte Reihenfolge zwingend vorgeschrieben. Aus dieser Forderung ergibt sich auch, daß jeder dieser Teile nur einmal definiert sein darf. Turbo Pascal ist hier wesentlich flexibler und schreibt lediglich zwei Dinge vor:

- Wenn eine **program**-Deklaration vorhanden ist, muß sie am Anfang des Quelltextes stehen.

- Jedes Programm muß ein Hauptprogramm als letzten Teil enthalten.

Zwischen der **program**-Deklaration und dem Hauptprogramm sind beliebige Folgen und Wiederholungen der anderen Teile erlaubt. (Die Grundregel von Pascal, daß *alle* Bezeichner vor ihrer Verwendung deklariert werden müssen, gilt aber auch in Turbo Pascal.)

Die Struktur einer Prozedur oder Funktion sieht eigentlich genauso aus wie die eines Programms. Fangen wir mit den Prozeduren an:

```
procedure <Name> (Parameter : Typ; Parameter : Typ ...);
label Labelname, Labelname...;
const Konstanten-Bezeichner = Wert; ...
type  Typ-Bezeichner = <Typ-Definition>;  ...
var   Variablen-Bezeichner : Typ-Bezeichner;  ....
```

```
    < Prozeduren und Funktionen >

begin
    < "Hauptprogramm" der Prozedur >
end.
```

Anstelle von **program** steht hier also das reservierte Wort **procedure**, auf den Namen der Prozedur folgen (optionale) Deklarationen von Parametern. Die restliche Struktur ist exakt gleich – auch innerhalb einer Prozedur können Labels, Typen, Variablen, Konstanten sowie weitere Prozeduren und Funktionen in beliebiger Folge deklariert sein. Den letzten Teil einer Prozedur stellt das eigentliche Programm dar, d.h. die Folge von Anweisungen, die beim Aufruf der Prozedur ausgeführt werden.

Funktionen unterscheiden sich von Prozeduren nur in zwei Punkten: Aus einer anderen Form der Kopfzeile, nämlich

```
function <Name> (Argument : Typ; Argument : Typ ...) : Ergebnistyp;
```

folgt offensichtlich, daß eine Funktion ein *Ergebnis* haben muß. Formulieren wir zur Demonstration unsere Übersetzung von Klein- auf Großbuchstaben als Funktion:

```
function SetUpcase(Ch: Char): Char;
begin
if (Ch >= 'a') and (Ch <= 'z')
   then Ch := chr(ord(ch) - 32)   { 'a' -> 'A', 'b'--> 'B'... }
   else if Ch = 'ä' then Ch := 'Ä'
   else if Ch = 'ö' then Ch := 'Ö'
   else if Ch = 'ü' then Ch := 'Ü'
   else ;   { kein Kleinbuchstabe --> keine Änderung }
   SetUpcase := Ch;   { legt das Ergebnis von SetUpcase fest }
end;
```

Die Deklaration *(Ch:Char): Char* bedeutet, daß dieser Funktion bei ihrem Aufruf eine Variable (oder ein Ausdruck) des Typs *Char* übergeben wird und daß sie einen Wert des Typs *Char* als *Funktionsergebnis* zurückliefert.

Das Funktionsergebnis wird durch die Zuweisung eines Wertes an den Funktionsnamen (in unserem Beispiel *SetUpcase :=Ch*) festgelegt, die innerhalb der Funktion stattfinden muß. Mehrere Zuweisungen an den Funktionsnamen sind möglich – wir hätten *SetUpcase* also auch so formulieren können:

```
function SetUpcase(Ch: Char): Char;
begin
if (Ch >= 'a') and (Ch <= 'z')
   then SetUpcase := chr(ord(ch) - 32)  { 'a' -> 'A', 'b'--> 'B'... }
   else if Ch = 'ä' then SetUpcase := 'Ä'
   else if Ch = 'ö' then SetUpcase := 'Ö'
   else if Ch = 'ü' then SetUpcase := 'Ü'
   else ;   { kein Kleinbuchstabe --> keine Änderung }
   SetUpcase := Ch;
end;
```

Der Aufruf innerhalb eines Programms erfolgt über den Funktionsnamen (in unserem Beispiel also *SetUpcase*) und kann an jeder Stelle stehen, an der auch eine Variable oder Konstante des Typs *Char* zulässig wäre. Einige Beispiele:

```
program SetUpcaseDemo;
var
  Eingabe, Ch: Char;
  ....     hier müßte die Funktion SetUpcase definiert sein
begin
  Readln(Eingabe);   { liest ein Zeichen }
  Writeln(Eingabe);
  Writeln(Eingabe, 'übersetzt in: ', SetUpcase(Eingabe));   { 1 }
  Eingabe := SetUpcase(Eingabe);                            { 2 }
  Ch := SetUpcase('ä');                                     { 3 }
end.
```

In der mit »1« kommentierten Zeile wird *SetUpcase* das Zeichen übergeben, das in der Variablen *Eingabe* gespeichert ist. Die Funktion liefert einen *Char* zurück, der von *Writeln* in derselben Weise ausgegeben wird wie eine andere *Char*-Variable oder Konstante. In der Zeile »2« bekommt *SetUpcase* wiederum den Wert von *Eingabe* übergeben und liefert ein Ergebnis zurück, das in *Eingabe* gespeichert wird – also in derselben Variablen, die wir auch als Funktionsargument benutzt haben. Möglich ist das deshalb, weil die rechte Seite einer Zuweisung erst komplett berechnet wird, bevor die Zuweisung tatsächlich erfolgt.

In der Zeile »3« übergeben wir *SetUpcase* schließlich eine Konstante des Typs *Char*, das Ergebnis wird der Variablen *Ch* zugeordnet. Moment! Haben wir diese Variable nicht schon innerhalb von *SetUpcase* selbst verwendet? Tatsächlich handelt es sich um zwei völlig voneinander getrennte Dinge – die Deklaration des Bezeichners *Ch* innerhalb von *SetUpcase* ist *lokal* zu dieser Funktion *und hat nichts mit der globalen Variablen Ch des Hauptprogramms zu tun*. Eine kurze Demonstration dieses Prinzips:

```
program funktioniert_nicht;
{var
   Ch: Char;   }     {  Deklaration als Kommentar versteckt }

procedure Nummer_1;
var
  Ch: Char;    { eine LOKALE Variable }
begin
  Writeln('Hier ist Prozedur Nummer_1');
  Ch := 'x';
end;

begin  { Hauptprogramm }
  Ch := 'A';
  Nummer_1;   { Aufruf der Prozedur Nummer_1 }
  Writeln(Ch);
end.
```

Wenn Sie dieses Programm compilieren, erhalten Sie die Meldung »Unbekannter Bezeichner« – eine Variable namens *Ch* ist auf der Ebene des Hauptprogramms nicht definiert. Entfernen Sie die Kommentarklammern um die Deklaration von *Ch* in der zweiten Zeile und probieren Sie es erneut. Ein Start des nun fehlerfrei compilierbaren Programms ergibt:

```
Hier ist die Prozedur Nummer_1
A
```

Die Variable *Ch* des Hauptprogramms ist also nicht verändert worden. Das Konzept der Lokalität gilt für *alle* Bezeichner, die innerhalb von Funktionen und Prozeduren deklariert werden, sowie für die Namen von Parametern bzw. Argumenten. Aus dem Hauptprogramm heraus kann *nie* auf lokale Bezeichner zugegriffen werden. Andersherum geht das dagegen schon: Entfernen Sie die Deklaration von *Ch* innerhalb der Prozedur *Nummer_1*, also:

```
program funktioniert_anders;
var
   Ch: Char;

procedure Nummer_1;
{ var
  Ch: Char;  }  {  Deklaration als Kommentar versteckt }
begin
  Writeln('Hier ist Prozedur Nummer_1');
  Ch := 'x';
  ....
```

und starten Sie das Programm erneut. Sie erhalten nun:

```
Hier ist die Prozedur Nummer_1
x
```

Prozeduren und Funktionen können weitere Prozeduren und Funktionen enthalten, die ihrerseits wieder lokale Variablen, Typen und Konstanten enthalten. Ein Beispiel:

```
program Verschachtelt;
var
  Ch: Char;
procedure Aussen(Ch: Char);
   procedure Innen(Ch: Char);
   begin
     Writeln('Hier Prozedur Innen: ', Ch);
     Ch := 'x';
   end;
begin { der Prozedur Aussen }
  Writeln('Hier Prozedur Aussen: ', Ch);
  Innen(Ch);
  Writeln('Noch einmal Prozedur Aussen: ', Ch);
  Ch := 'y';
end;
```

```
begin  { Hauptprogramm }
  Ch := 'A';
  Aussen(Ch);
  Writeln('Hauptprogramm: ', Ch);
end.
```

Die Ausgaben dieses Programms:

```
Hier Prozedur Aussen: A
Hier Prozedur Innen: A
Noch einmal Prozedur Aussen: A
Hauptprogramm: A
```

Die Prozedur *Innen* kann nur innerhalb der Prozedur *Aussen* aufgerufen werden. Sowohl *Aussen* als auch *Innen* verändern den übergebenen Parameter *Ch* – wie die Ausgaben des Programms beweisen, haben diese Veränderungen aber keine Rückwirkung auf die jeweils übergeordnete Ebene. Falls derartige Rückwirkungen erwünscht sind, muß der entsprechende Parameter mit dem Zusatz **var** deklariert werden:

```
procedure Parameter_Aendern(var Ch: Char);
begin
  Ch := 'x';
end;
```

Ein Aufruf dieser Prozedur, beispielsweise mit

```
Parameter_Aendern(Eingabe);
```

übergibt nicht den *Wert*, der in der Variablen *Eingabe* gespeichert ist, sondern einen Zeiger auf die Speicherzellen, in denen *Eingabe* tatsächlich gespeichert ist. Demzufolge spricht man hier auch nicht von einer *Wertübergabe*, sondern von einer *Variablenübergabe*. Kapitel 8 »Prozeduren und Funktionen« des *Programmierhandbuches* beschäftigt sich eingehender mit den Unterschieden beider Formen. Die hier erläuterten Prinzipien sind übrigens in gleicher Weise auf Funktionen anwendbar.

Kommentare

sind zusätzliche Erläuterungen zur Funktion von Variablen, Anweisungen und zentralen Punkten eines Programms. Sie sind außerordentlich wichtig – nicht nur für andere, die eventuell damit zurechtkommen müssen, sondern auch für Sie, wenn Sie eines Ihrer Programme nach einigen Monaten noch einmal überarbeiten wollen.

Ein Kommentar wird in Pascal mit einer öffnenden geschweiften Klammer ({) eingeleitet und mit einer schließenden geschweiften Klammer (}) beendet. Sämtliche Zeichen zwischen diesen beiden Folgen werden vom Compiler ignoriert und haben keinen Einfluß auf das Ergebnis der Übersetzung.

Kommentare können sich also auch über mehrere Zeilen hinweg erstrecken:

```
{ Das ist ein langer
 Kommentar, der über drei
      Zeilen geht }
```

Alternativ zu geschweiften Klammern erlaubt Turbo Pascal die Verwendung der Zeichenfolgen (* und *). Beide Formen dürfen innerhalb *eines* Kommentars nicht gemischt werden – ein Kommentar, der mit einer geschweiften Klammer beginnt, darf also nicht mit *) aufhören und umgekehrt.

Verschachtelte Kommentare sind im Standard-Pascal nicht erlaubt. Ein Versuch wie der folgende führt zu einer Fehlermeldung des Compilers:

```
{  dieser Programmteil soll ausgeklammert werden
Writeln; { ein Kommentar }     hier hält der Compiler den
                               gesamten "Kommentar" für beendet
Writeln; { noch ein Kommentar }   wird compiliert
}   erzeugt eine Fehlermeldung
```

In Turbo Pascal ist das »Hinauskommentieren« von Programmteilen durch die Unterscheidung zwischen den Zeichenfolgen (* und *) und geschweiften Klammern möglich – allerdings nur, solange der Programmteil durchgehend dieselbe Art von Kommentaren enthält:

```
(*  dieser Programmteil soll ausgeklammert werden
Writeln; { ein Kommentar }
Writeln; { noch ein Kommentar }
*)      diese Art der Ausklammerung ist zulässig

{  dieser Programmteil soll ausgeklammert werden
Writeln; (* ein Kommentar *)
Writeln; (* noch ein Kommentar *)
}      diese Art der Ausklammerung ist ebenfalls möglich
```

Das achte Element: Fehlersichere Programme

So – nun haben wir die nötigen Grundlagen, um uns wie angekündigt um das rätselhafte Ende des Programms *Nummer3* in Kapitel 1 zu kümmern. Kurz noch einmal zur Erinnerung: Ein Aufruf von *Readln* mit

```
Readln(A, B);    { A und B sind Integer-Variablen }
```

bricht das Programm mit einer Fehlermeldung ab, wenn nicht-numerische Zeichen (z.B. *A5* anstelle von *35*) eingegeben werden. In einem kleinen Programm wie *Nummer3* ist ein plötzlicher Abbruch wegen eines Tippfehlers kein Beinbruch – aber stellen Sie sich einmal vor, Sie hätten eine Datenbank vor sich und gerade einige Hundert Werte eingegeben und dürften nun von vorne anfangen! Eine hieb- und stichfeste Eingabe ist eine der vielen Voraussetzungen, die ein verwendbares Programm erfüllen muß.

Wie die meisten anderen Compiler, erlaubt auch Turbo Pascal das zeitweilige Abschalten der automatischen Fehlerprüfung für Ein- und Ausgaben – entsprechende Tests können Sie dann selbst innerhalb des Programms vornehmen. Mit `{$I-}` wird die Überprüfung abgeschaltet, durch den Befehl `{$I+}` kann sie zu

einem späteren Zeitpunkt wieder aktiviert werden. Nur im Modus {$I+} erzeugt der Compiler zusätzlichen Prüfcode, der das Programm bei ungültigen Eingaben abbricht. Eine erste (und sehr primitive) Modifikation des Programms *Nummer3* könnte in etwa so aussehen:

```
programNummer3;       { erste Erweiterung }
var
  A, B: Integer;
  C: Integer;

begin
  Write ('Geben Sie zwei Werte ein: ');
  {$I-}                { schaltet die Überprüfung ab }
  Readln (A, B);
  {$I+}                { Überprüfung wieder an }
  if B <> 0 then
   begin     { verhindert eine mögliche Division durch 0 }
     C:= A div B;
     Writeln ('Das Ergebnis ist: ', C);
   end;
end.
```

Bei diesem Programm spielt es keine Rolle, was Sie für A und B eingeben – selbst nicht-numerische Zeichenfolgen erzeugen keine Fehlermeldungen. Was natürlich nicht heißen soll, daß sich das Programm unter allen Umständen vernünftig verhält: nicht auswertbare Eingaben setzen einfach die entsprechende Variable auf den Wert 0.

Solange die Überprüfung der Ein-/Ausgabe mit {$I-} abgeschaltet ist, können Sie durch den Aufruf der Standardfunktion *IOResult* selbst prüfen, ob sich ein Fehler ereignet hat. *IOResult* liefert einen Integerwert zurück, der dem Status der jeweils letzten Ein-/Ausgabe entspricht (vgl. Kapitel 3 »Fehlermeldungen und -codes« des *Referenzhandbuches*). Der Wert 0 als Ergebnis von *IOResult* bedeutet, daß die jeweils letzte Ein- oder Ausgabeoperation fehlerfrei verlaufen ist. Bei anderen Ergebnissen sollte das Programm entsprechend reagieren, und sei es nur mit einer Wiederholung der Eingabeanforderung:

```
programNummer3;       { zweite Erweiterung }
var
  A, B, C: Integer;

begin
  {$I-}                { schaltet die Überprüfung ab }
  repeat
    Write ('Geben Sie zwei Werte ein: ');
    readln (A, B);
  until IOResult = 0;
  {$I+}                { Überprüfung wieder an }
  if B <> 0 then
   begin  { verhindert eine mögliche Division durch 0 }
     C:= A div B;
```

```
      Writeln ('Das Ergebnis ist: ', C);
    end;
end.
```

Achtung: Jeder Aufruf von *IOResult* liefert den jeweils letzten Fehlercode und setzt dabei eine eventuelle Fehlerbedingung automatisch zurück. Außerdem wird *IOResult* durch jede Ein-/Ausgabe neu gesetzt, also auch durch *Write*, *Writeln*, *Read* und *Readln*, *Assign*, *Reset*, *Rewrite* usw. Aus diesem Grund kann die folgende Version von *Nummer3* nicht in der gewünschten Weise funktionieren:

```
programNummer3;  { dritte Version - funktioniert nicht! }
var
  A, B, C: Integer;
begin
  {$I-}                { schaltet die Überprüfung ab }
  repeat
    Write('Geben Sie zwei Werte ein: ');
    Readln (A, B);
    if IOResult <> 0 then
       Writeln ('Fehlerhafte Eingabe!');
  until IOResult = 0;
  {$I+}                { Überprüfung wieder an }
  if B <> 0 then
   begin  { verhindert eine mögliche Division durch 0 }
     C:= A div B;
     Writeln ('Das Ergebnis ist: ', C);
   end;
end.
```

Es gibt zwei Gründe aus denen dieses Programm bei fehlerhaften Eingaben mit ungültigen Werten für A und B weiterrechnet: Zum einen liefert der Aufruf von *IOResult* durch die Anweisungsfolge **if** *IOResult* <>*0 ...* zwar den Fehlerstatus der Eingabe durch *Readln* zurück, löscht aber eine eventuelle Fehlerbedingung – die **until**-Anweisung am Ende der Schleife wird also beim zweiten Aufruf von *IOResult* immer das Ergebnis »kein Fehler« erhalten. Zum anderen setzt die Ausgabe der Fehlermeldung (via *Writeln*) *IOResult* ebenfalls neu – in diesem Fall wertet der Aufruf von *IOResult* bei **until** aus, ob während der Ausführung von *Writeln* ein Fehler aufgetreten ist.

Wenn Sie eine Fehlermeldung auf den Bildschirm ausgeben wollen, müssen Sie eine Hilfsvariable benutzen:

```
programNummer3;    { Version 4, mit Hilfsvariable }
var
  A, B, Fehlercode: Integer;
  C   : Integer;
begin
  {$I-}                { schaltet die Überprüfung ab }
  repeat
    Write('Geben Sie zwei Werte ein: ');
    Readln (A, B);
```

```
    Fehlercode:= IOResult
    if Fehlercode <> 0 then
       Writeln ('Fehlerhafte Eingabe!');
  Until Fehlercode = 0;
  {$I+}                { Überprüfung wieder an }
  if B <> 0 then
   begin  { verhindert eine mögliche Division durch 0 }
     C:= A div B;
     Writeln ('Das Ergebnis ist: ', C);
   end;
end.
```

Indem wir das Ergebnis von *IOResult* in der Variablen Fehlercode speichern, umgehen wir beide Probleme: *IOResult* wird hier nur einmal benutzt, Fehlercode enthält während der gesamten Schleife den Status von *Readln* und wird durch weitere Ausgaben auf den Bildschirm nicht verändert.

3. Units, Overlays und andere Geheimnisse

Nachdem wir uns nun ausführlich mit den Grundlagen der Programmiersprache Pascal beschäftigt haben, sollte das Schreiben kleinerer Standard-Programme kein allzugroßes Problem mehr darstellen. Wie aber sieht es mit Programmen aus, die über den Sprachstandard hinausgehen: mit direkter Kontrolle des Bildschirms, Grafik, und Aufrufen von DOS? Um derartige Programme zu schreiben, müssen Sie sich entweder recht gut mit der Elektronik Ihres Computers auskennen – oder Sie benutzen die Standard-Units von Turbo Pascal, in denen Routinen für fast alles und jedes vordefiniert sind.

Dieses Kapitel klärt zuerst, woraus eine Unit besteht, wie man sie benutzt und welche Standard-Units Sie zur Verfügung haben. Nach der Erläuterung, wie man eigene Units erstellt, und einem kurzen Ausflug zum Thema »modulare Programmierung« kommen wir zu einem weiteren Konzept von Turbo Pascal, in dem Units ebenfalls eine wichtige Rolle spielen: der Arbeit mit Overlays.

Was bitte ist eine Unit?

Turbo Pascal deklariert eine sehr große Zahl von Konstanten, Datentypen, Variablen, Prozeduren und Funktionen (insgesamt dürfen es rund tausend Bezeichner sein). Ein Teil davon – der Standard-Sprachumfang – gehört direkt zur Laufzeitbibliothek, ein anderer ist spezifisch für den IBM PC bzw. MS-DOS geschrieben und in separaten Units gespeichert. Dieser Teil steht erst dann zur Verfügung, wenn die entsprechende Unit in ein Programm aufgenommen wird.

Jede Unit ist im Grunde fast ein selbständiges Pascal-Programm: sie enthält Konstanten, Datentypen, Variablen, Prozeduren und Funktionen. Sogar ein eigenes Hauptprogramm ist möglich, das vor dem eigentlichen Programmstart aufgerufen wird und Initialisierungen vornimmt. Kurz gesagt ist eine Unit eine Bibliothek von Deklarationen, die separat compiliert, bei Bedarf in ein Programm aufgenommen und dort benutzt werden können.

Alle Deklarationen innerhalb einer Unit stehen normalerweise in einem Zusammenhang. So enthält beispielsweise die Unit *Crt* sämtliche Deklarationen für die direkte Ansteuerung des (Text-)Bildschirms.

Turbo Pascal stellt acht Standard-Units zur Verfügung, von denen fünf (*System*, *Crt*, *Dos*, *Printer* und *Overlay*) in der Datei TURBO.TPL gespeichert sind.

Diese Units stellen zusammen mit der Datei GRAPH.TPU (welche die Unit *Graph* enthält) sozusagen den Kern der Version 6.0 dar. Die letzten beiden – *Turbo3* und *Graph3* – erleichtern die Übernahme existierender Programme der Version 3.0 und existieren wie GRAPH.TPU als separate Dateien.

Hinweis: Turbo Vision stellt zusätzliche Units bereit. Lesen Sie dazu das Handbuch *Objektorientierte Programmierung und Turbo Vision*.

Anhang A »Unterschiede der Versionen« setzt sich speziell mit der Konvertierung existierender Programme (und mit diesen beiden »Kompatibilitäts-Units«) auseinander, Kapitel 12 »Standard-Units« des *Programmierhandbuches* beschreibt die Routinen aller acht Standard-Units in tabellarischer Form.

Zur Struktur

Eine Unit besteht aus drei Teilen: dem *Interface* (der Schnittstelle), der *Implementation* (d.h. dem tatsächlichen Programmcode) und der *Initialisierung* (einer Art Hauptprogramm). Der Interface-Teil ist der »öffentliche« Teil mit den Definitionen der Datentypen, Variablen, Konstanten sowie einer Auflistung der verfügbaren Funktionen und Prozeduren. Der Implementations-Teil ist dagegen »privat«: er enthält die Blöcke der Prozeduren und Funktionen, die im Interface deklariert worden sind. Außerdem können Sie hier zusätzliche Konstanten, Datentypen, Variablen und Routinen deklarieren, die nur innerhalb der Implementation benutzt werden und nicht »öffentlich« verfügbar sind. Der Initialisierungs-Teil wird von Turbo Pascal vor dem Start des Programms aufgerufen, das diese Unit benutzt. Wie der Name schon andeutet, können hier Initialisierungen von Datenstrukturen und einleitende Schritte ausgeführt werden (wie beispielsweise das Öffnen der Datei *Lst* durch die Unit *Printer*).

Wie bereits gesagt, ist die Struktur einer Unit durchaus mit der eines Programms vergleichbar. Ein erster Blick auf die Struktur zeigt nur wenige Unterschiede:

```
unit <Name>;
interface
  uses <Liste "öffentlich" benutzter Units>;  { optional }
  { öffentliche Deklarationen }
implementation
  uses <Liste "privat" benutzter Units>;  { optional }
  { nicht-öffentliche Deklarationen }
  { Prozeduren und Funktionen }
begin
  {Initialisierung}
end.
```

Der Kopf der Unit beginnt mit dem reservierten Wort **unit**, gefolgt von ihrem Namen. Der Name der Unit hat denselben Zweck wie der eines Programms: Er dient zur Identifikation (und ist normalerweise mit dem Dateinamen des Unit-Quelltextes identisch). Auf die Kopfzeile folgt das reservierte Wort **interface**, das den Beginn der »öffentlichen« Deklarationen kennzeichnet.

Eine Unit kann andere Units voraussetzen und muß sie in diesem Fall mit **uses** aufführen. Im Gegensatz zu einem Programm – das nur eine einzige **uses**-Anweisung enthalten darf – gibt es hier zwei verschiedene Möglichkeiten:

- eine **uses**-Anweisung, die direkt auf das reservierte Wort **interface** folgt, hat dieselbe Wirkung wie eine **uses**-Anweisung innerhalb eines Programms: die Deklarationen damit aufgenommener Units werden global (innerhalb dieser Unit) verfügbar;
- eine **uses**-Anweisung, die direkt auf das reservierte Wort **implementation** folgt, macht die Deklarationen damit aufgenommener Units nur »privat« (d.h. im Implementations-Teil dieser Unit) verfügbar.

Etwas kürzer und anhand eines Beispiels: Eine Unit mit einer Struktur wie

```
unit x;
interface
uses y;
implementation
uses z;
end.
```

kann die Deklarationen der Unit *y* sowohl in seinem Interface-Teil als auch im Implementations-Teil benutzen; die Deklarationen der Unit *z* lassen sich dagegen nur im Implementations-Teil von *x* verwenden. Diese Trennung macht Units möglich, die sich gegenseitig voraussetzen – ein Thema, auf das wir später noch zurückkommen.

Interface

Der Interface-Teil einer Unit beginnt mit dem reservierten Wort **interface**, das unterhalb der Kopfzeile steht und mit dem reservierten Wort **implementation** endet. Innerhalb des Interface wird festgelegt, welche Bestandteile der Unit für ein Programm oder andere Units zugänglich sind. Im Interface-Teil können Konstanten, Datentypen, Variablen, Funktionen und Prozeduren deklariert werden. Genauso wie innerhalb eines Programms, lassen sich diese Deklarationen beliebig anordnen, die Wiederholung einzelner Deklarationsteile ist ebenfalls möglich:

```
type... var... <Prozedur-Kopf>... const... type... var
```

Wenn ein Programm eine Unit benutzt, werden dadurch *alle* Deklarationen des Interface-Teils dieser Unit verfügbar – genauso, als hätte man sie direkt in den Quelltext hineingeschrieben.

Prozeduren und Funktionen, die von außerhalb der Unit aufgerufen werden können, werden ebenfalls im Interface-Teil deklariert – allerdings nur als Kopf: der eigentliche Code befindet sich im Implementations-Teil. Um eine Funktion oder Prozedur als **external** zu deklarieren, müssen Sie den entsprechenden Kopf, gefolgt von dem reservierten Wort **external**, im Interface-Teil aufführen – eine weitere Deklaration im Implementations-Teil gibt es nicht. Ähnliches gilt für **inline**: Der Funktions- bzw. Prozedurkopf steht zusammen mit der als **inline**

angegebenen Bytefolge innerhalb des Interface-Teils. **forward**-Deklarationen sind innerhalb einer Unit weder notwendig noch zulässig.

Wenn der Implementationsteil einer Unit eine **uses**-Anweisung enthält, muß sie direkt auf das Schlüsselwort **implementation** folgen.

Implementation

Dieser Teil beginnt mit dem reservierten Wort **implementation** – hier befinden sich die Blöcke aller »regulären« Funktionen und Prozeduren. Alle Deklarationen des Interface-Teils (d.h. Konstanten, Datentypen und Variablen und Routinen) sind für die im Implementations-Teil aufgeführten Programmstücke verfügbar.

Zusätzliche Deklarationen innerhalb der Implementation sind für das Programm unsichtbar, das diese Unit benutzt. Das Programm »weiß« weder von ihrer Existenz, noch kann es sie benutzen – das einzige, was dem Programm zur Verfügung steht, sind die Definitionen innerhalb des Interface.

Wenn Prozeduren oder Funktionen innerhalb des Interface-Teils als **external** deklariert sind, müssen die dazugehörigen Compiler-Befehle ({$L *OBJ.Dateiname*}) vor dem **begin** des Initialisierungs-Teils programmiert werden – also innerhalb der Implementation! Falls die Unit keinen Initialisierungs-Teil definiert, dann können diese Compiler-Befehle an eine beliebige Stelle (vor dem abschließenden **end**) gesetzt werden.

Alle »regulären« Funktionen und Prozeduren (also weder **inline** noch **external**), die innerhalb des Interface deklariert worden sind, müssen im Implementations-Teil erneut erscheinen, wobei entweder

- nur das reservierte Wort **procedure** bzw. **function** zusammen mit dem Namen der Routine angegeben wird, oder
- der gesamte Kopf der Routine (inklusive aller Parameter) erneut deklariert ist.

Der Compiler prüft in diesem Fall, ob die Angaben in Interface und Implementation übereinstimmen und führt sonst keine zusätzlichen Schritte aus.

Auf den Kopf der Routine folgen lokale Deklarationen (Konstanten, Variablen, Datentypen, verschachtelte Funktionen, Prozeduren und Labels) sowie der Block mit den eigentlichen Befehlen.

Nach so viel Theorie ein praktisches Beispiel. Wenn der Interface-Teil einer Unit die folgenden Deklarationen enthält:

```
procedure ISwap(var V1,V2: Integer);
function IMax(V1,V2: Integer): Integer;
```

dann könnte eine entsprechende Implementation so aussehen:

```
procedure ISwap;
var Temp: Integer;
begin
  Temp:= V1; V1:= V2; V2:= Temp
end;  {der Prozedur ISwap}

function IMax;
begin
```

```
  if V1 > V2 then
     IMax:= V1
  else IMax:= V2
end;  {der Funktion IMax}
```

Routinen, die lokal innerhalb der Implementation definiert sind (also nicht innerhalb des Interface erscheinen), werden genauso deklariert und ausgeführt wie in einem normalen Programm: der Kopf wird nur einmal zusammen mit eventuellen Parametern angegeben, darauf folgt der Block.

Initialisierung

Normalerweise wird der Implementations-Teil einer Unit von den reservierten Wörtern **implementation** und **end** eingeschlossen. Es ist allerdings möglich, einer Unit ein »Hauptprogramm« zu geben: Fügen Sie nach dem letzten Prozedur-/Funktionsrumpf des Implementations-Teils das Wort **begin** ein. Auf diese Einleitung kann (wie auch im Hauptprogramm eines normalen Pascal-Programms) eine beliebige Anzahl von Anweisungen folgen, die durch *dasselbe* **end** wie normalerweise der Implementations-Teil abgeschlossen werden. Beim Start eines Programms, das diese Unit benutzt, wird nun zuerst dieser Initialisierungs-Teil ausgeführt – wenn mehrere Units benutzt werden, dann geschieht der Aufruf eventuell definierter Initialisierungs-Teile in der Reihenfolge, in der die Units mit **uses** angegeben sind.

Hinweis: Units, die ihrerseits weitere Units aufnehmen, setzen diese Regel außer Kraft (siehe nächster Abschnitt).

Innerhalb des Initialisierungs-Teils einer Unit sind beliebige Operationen möglich, die sowohl »öffentliche« als auch »nicht-öffentliche« Datenstrukturen mit bestimmten Werten setzen und/oder Routinen der Unit aufrufen. Hier können auch Dateien geöffnet werden, die das Programm später verwendet: Die Unit *Printer* verwendet beispielsweise ihren Initialisierungs-Teil, um die Textdatei *Lst* für Ausgaben zu öffnen, und ermöglicht so dem Programm Ausgaben auf einen angeschlossenen Drucker über Aufrufe von *Write* und *Writeln*.

Wie benutzt man eine Unit?

Alle durch Turbo Pascal zur Verfügung gestellten Units sind bereits compiliert, d.h. sie sind nicht als Quelltext, sondern in Form von Maschinencode gespeichert. Units sind also *keine* Include-Dateien! Selbst der Interface-Teil ist nicht im ASCII-Format, sondern in einem speziellen binären Format von Turbo Pascal gehalten.

Die fünf wichtigsten vordefinierten Units sind Teil der Datei TURBO.TPL und werden von Turbo Pascal beim Start der integrierten Entwicklungsumgebung bzw. der Kommandozeilen-Version des Compilers automatisch geladen. Der Vorteil dieser Speicherform: Die Aufnahme von Units in ein Programm besteht lediglich aus einer Einbindung (also keiner Compilierung) und kostet deshalb sehr wenig Zeit. Wenn Sie Units von einer eigenen Diskettendatei laden, müssen Sie dagegen einige zusätzliche Sekunden Compilierungszeit in Kauf nehmen.

Um eine oder mehrere Units in einem Programm zu benutzen, müssen Sie am Beginn Ihres Programms die Anweisung **uses** zusammen mit einer Liste der entsprechenden Units einfügen. Mehrere Unit-Namen werden durch Kommatas voneinander getrennt, wie das folgende Beispiel zeigt:

```
program UnitDemo;
uses EineUnit, NocheineUnit, EinedritteUnit;
```

Sobald der Compiler auf eine **uses**-Anweisung trifft, trägt er die entsprechenden Interface-Deklarationen in seine Symboltabelle ein und stellt dem Linker den Maschinencode des Implementations-Teils zur Verfügung.

Die Verwaltung von Units, die ihrerseits von anderen Units vorausgesetzt werden, geschieht automatisch: Wenn eine Unit wie

```
unit UnitA;
interface
uses UnitB;
  ...
```

in ein Programm mit

```
uses UnitA;
```

aufgenommen wird, dann bindet der Linker die von *UnitA* vorausgesetzte *UnitB* automatisch mit ein. Dieses Einbinden erfolgt sozusagen **vor** *UnitA* – schließlich könnte es sein, daß *UnitA* einen Initialisierungs-Teil enthält, der auf Initialisierungen von *UnitB* aufbaut. Anders gesagt: Für unser Beispiel verhält sich der Compiler so, als ob das Hauptprogramm die Anweisung

```
uses UnitB, UnitA;
```

enthalten würde, und wendet dann die Regel an, daß die Initialisierungs-Teile der Units in der Reihenfolge ihrer Angabe mit **uses** aufgerufen werden. Wenn *UnitA*, *UnitB* und das Hauptprogramm jeweils eine Anweisung der Form

```
Writeln('Hier ist ... ');
```

in ihren Initialisierungs-Teilen bzw. direkt nach dem **begin** des Hauptprogramms enthalten, dann erzeugt das Programm beim Start die folgenden Ausgaben:

```
Hier ist <UnitB>
Hier ist <UnitA>
Hier ist <Hauptprogramm>
```

Die Deklarationen von *UnitB* sind in diesem Beispiel nur innerhalb von *UnitA* verfügbar – wenn sie auch vom Hauptprogramm benutzt werden sollen, muß das Hauptprogramm selbst eine entsprechende **uses**-Anweisung enthalten, die *UnitB* explizit aufführt. Da *UnitA* nach wie vor *UnitB* voraussetzt, spielt die Reihenfolge der Angabe auch in diesem Fall eine untergeordnete Rolle (weil sie vom Compiler nötigenfalls selbständig festgelegt wird):

```
program Unit1Unit2;
uses UnitA, UnitB;
```

```
program Unit2Unit1;
uses UnitB, UnitA;
```

Diese beiden Deklarationsfolgen sind in unserem Beispiel für Compiler und Linker identisch.

Hinweis: Der Linker ist intelligent genug, um vorausgesetzte Units grundsätzlich nur *einmal* in das gesamte Programm aufzunehmen – unabhängig davon, von wievielen Modulen des Programms diese Unit mit **uses** aufgeführt wird.

Selbst wenn Sie überhaupt keine **uses**-Anweisung verwenden, bindet Turbo Pascal die Unit *System* in Ihr Programm mit ein. *System* deklariert alle Routinen, Variablen und Datentypen des Standard-Pascal sowie einige Turbo-spezifische Erweiterungen – stellt also die Bezeichner zur Verfügung, auf die auch in einem Standard-Programm zugegriffen werden kann.

Die Standard-Units

Die Datei TURBO.TPL enthält fünf der acht Standard-Units (*System*, *Dos*, *Crt*, *Overlay* und *Printer*) von Turbo Pascal und wird beim Start der integrierten Entwicklungsumgebung bzw. der Kommandozeilen-Version automatisch geladen. Sie sollte deshalb im selben Verzeichnis wie TURBO.EXE bzw. TPC.EXE gespeichert sein.

Die folgende Aufstellung soll nur einen kurzen Überblick vermitteln – Details zu einzelnen Routinen und Bezeichnern finden Sie in Kapitel 12 »Standard-Units« des *Programmierhandbuches*.

System (vorausgesetzte Units: keine): Diese Unit enthält die Grundfunktionen von Turbo Pascal und wird automatisch in jedes Programm aufgenommen. Eine entsprechende **uses**-Anweisung ist weder notwendig noch erlaubt.

Dos (vorausgesetzte Units: keine): *Dos* ist die Schnittstelle zum Betriebssystem. Diese Unit deklariert Prozeduren, Funktionen und Datentypen, über die ein Pascal-Programm Funktionen von MS-DOS (wie *GetTime*, *SetTime*, *DiskSize* usw.) aufrufen kann. Außerdem finden sich hier die beiden Prozeduren *MsDos* und *Intr* zusammen mit dem Record-Typ *Registers*, die das direkte Auslösen von Interrupts ermöglichen.

Crt (vorausgesetzte Units: keine): *Crt* ermöglicht die direkte Kontrolle des (Text-)Bildschirms, der Tastatur und des eingebauten Lautsprechers. In dieser Unit sind Konstanten für Bildschirmfarben, die Prozedur *Window* und Routinen zur Steuerung des Cursors deklariert.

Overlay (vorausgesetzte Units: keine): Wie der Name vermuten läßt, enthält *Overlay* die Overlay-Verwaltung von Turbo Pascal, mit der wir uns im zweiten Teil dieses Kapitels noch ausführlich auseinandersetzen.

Printer (vorausgesetzte Units: keine): *Printer* deklariert die Textdatei-Variable *Lst* und verbindet sie mit einem Gerätetreiber, der Ausgaben auf einen Drucker mit *Write* und *Writeln* ermöglicht.

Graph (vorausgesetzte Units: keine): *Graph* ist ein komplettes Grafikpaket und arbeitet (fast) unabhängig vom verwendeten Video-Adapter. In dieser Unit sind nicht nur Routinen für das Zeichnen von Linien und Punkten, Kuchendiagrammen und Kreisen enthalten, sondern auch für die grafische Ausgabe von Text. Die folgenden Video-Adapter werden unterstützt: CGA, MCGA, EGA, VGA, Hercules, AT&T 400, 3270 PC und IBM 8514.

Hinweis: Graph ist als eigenständige Datei (GRAPH.TPU) gespeichert und sollte von INSTALL zusammen mit Treibern (*.BGI) bzw. Zeichensätzen (*.CHR) in dasselbe Verzeichnis wie TURBO.EXE kopiert worden sein.

Turbo3 (vorausgesetzte Units: *Crt*): Diese Unit erleichtert die Übernahme existierender Programme durch die Deklaration verschiedener Bezeichner, die seit der Version 4.0 nicht mehr definiert sind oder eine andere Arbeitsweise haben.

Graph3 (vorausgesetzte Units: *Crt*): *Graph3* ist ebenfalls für die Übernahme existierender Programme vorgesehen und enthält sämtliche Grafik-Funktionen der Version 3.0 (inklusive der Turtlegrafik).

Erweiterungen von TURBO.TPL: Das Programm TPUMOVER

Standard-Units wie *Graph*, *Turbo3* und *Graph3* müssen übrigens nicht unbedingt als einzelne Dateien gespeichert sein: Mit TPUMOVER – einem eigenständigen Programm – können Sie diese Units zu TURBO.TPL hinzufügen und so dafür sorgen, daß sie bei jedem Start von TURBO.EXE bzw. TPC.EXE automatisch geladen werden.

Natürlich ist TPUMOVER weder auf TURBO.TPL als Bibliothek noch auf Standard-Units beschränkt – auch eigene Units lassen sich mit diesem Programm zum Sprachumfang von Turbo Pascal hinzufügen. Die technischen Details dazu finden Sie in Kapitel 10 »Die Zusatzprogramme«.

Selbstgeschriebene Units

Bis auf die noch offene Frage, wozu **uses**-Anweisungen im Implementations-Teil gut sind, haben wir die Struktur von Units nun recht gründlich geklärt – und eine selbstgeschriebene Unit unterscheidet sich in keinem Punkt von den Standard-Units, die bereits in Turbo Pascal enthalten sind. Fangen wir mit einem Beispiel an:

```
unit IntLib;     { Name der Unit: IntLib }
interface        { "öffentliche" Deklarationen }

  procedure ISwap (var I,J: Integer);
  function IMax (I,J: Integer): Integer;

implementation { Ausführung der Prozeduren/Funktionen }
```

```
  procedure ISwap;
    var
      Temp: Integer;     { nur innerhalb von ISwap verfügbar }
    begin
      Temp:= I; I:= J; J:= Temp;
    end;   { Ende der Prozedur ISwap }

  function IMax;
    begin
      if I > J then IMax:= I
               else IMax:= J;
    end;   {Ende der Funktion IMax}

      {<- Hier steht kein begin: kein Initialisierungs-Teil}
end.  {Ende der Unit IntLib }
```

Geben Sie dieses Programm ein und speichern Sie es als INTLIB.PAS. Die Compilierung geschieht in derselben Weise wie bei einem normalen Programm (d.h. mit Alt C C oder F9 innerhalb der integrierten Entwicklungsumgebung). Da der Quelltext aber nicht mit dem reservierten Wort **program**, sondern mit **unit** eingeleitet ist, bekommt die erzeugte Datei bei der Compilierung auf die Diskette die Extension .TPU (»Turbo Pascal Unit«) und stellt kein ausführbares Programm dar.

Hinweis: Die .TPU-Datei wird in dem über *O/D/EXE & TPU directory* gesetzten Verzeichnis gespeichert. Damit Turbo Pascal INTLIB.TPU bei der späteren Aufnahme in ein Programm findet, muß dieses Verzeichnis entweder als Standard gesetzt oder in den über *O/D/Unit directories* angegebenen Suchwegen enthalten sein. Das folgende Programm benutzt die in INTLIB deklarierten Routinen:

```
program IntTest;
uses IntLib;

var
  A, B: Integer;

begin              {gleich mit dem Hauptprogramm}
  Write ('Geben Sie zwei Integerwerte ein: ');
  Readln (A, B);

  ISwap (A, B);    {diese Prozedur ist aus IntLib!}

  Writeln ('Das Maximum ist ', IMax(A,B));
end.               {Ende des Programms IntTest}
```

Einige Formalitäten

Wir haben uns im vorangegangenen Beispiel nicht allzusehr um Formales gekümmert – natürlich gibt es auch für die Erstellung eigener Units und ihre Aufnahme in ein Programm diverse Regeln, die eingehalten werden müssen.

- Mit jeder Quelltext-Datei kann nur eine Unit erzeugt werden. Wie bei einem normalen Pascal-Programm beendet der Compiler seine Arbeit mit dem abschließenden **end.** des Initialisierungs-Teils.

- Der Name der .TPU-Datei ergibt sich aus dem Namen des Quelltextes. Die Compilierung der Datei MYUNIT.PAS ergibt also die Datei MYUNIT.TPU – unabhängig davon, welchen Namen Sie innerhalb dieser Datei mit **unit** deklariert haben.

- Der Compiler sucht bei der Aufnahme einer Unit in ein Programm zuerst die Datei TURBO.TPL ab (in die sich mit TPUMOVER auch selbstgeschriebene Units aufnehmen lassen); wenn sich die gesuchte Unit hier nicht befindet, wird nach einer .TPU-Datei gesucht, die denselben Namen wie die Unit hat. Der Quelltext einer Unit wie *IntLib* muß in diesem Fall also in der Datei INTLIB.PAS gespeichert (und als INTLIB.TPU compiliert) worden sein.

- Wenn der Compiler über *C/Compile* aufgerufen wird und eine mit **uses** angegebene Unit nicht als .TPU-Datei findet, bricht er mit einer entsprechenden Fehlermeldung ab; eine Prüfung, ob ein Quelltext für die Unit vorhanden ist, findet nicht statt. Beim Aufruf über *C/Make* wird der Unit-Quelltext nötigenfalls compiliert – und zwar unabhängig vom Stand des Schalters *C/Destination* als .TPU-Datei auf die Diskette. Zur Verdeutlichung dieses Sachverhalts greifen wir noch einmal auf unser Beispiel mit *IntLib* und *IntTest* zurück: Wenn INTLIB.TPU nicht vorhanden ist, bricht die Compilierung von *IntTest* über *C/Compile* mit einer Fehlermeldung ab; bei einer Compilierung über *C/Make* oder F9 wird dagegen INTLIB.PAS gesucht, als INTLIB.TPU compiliert und anschließend in das Programm *IntTest* aufgenommen. Die Kapitel 5 »Verwaltung von Projekten« und 10 »Die Zusatzprogramme« erläutern die zugrundeliegende Logik im Detail.

Units und mehrfach deklarierte Bezeichner

Wie bereits erläutert, macht die Aufnahme einer Unit in ein Programm alle Bezeichner global verfügbar, die im Interface-Teil dieser Unit deklariert sind. Was passiert, wenn Sie in Ihrem Programm eine Variable, Konstante oder Routine deklarieren, die innerhalb einer aufgenommenen Unit bereits vorhanden ist? Bevor wir auf die theoretischen Hintergründe eingehen, erst einmal ein Beispiel zur Verdeutlichung:

```
program Mehrfach;
uses
   Crt,     { deklariert Farbkonstanten für Text }
   Graph;    { deklariert Farbkonstanten für Grafik }

const
   Black = 0;  { eine Deklaration innerhalb des Programms }

procedure xxx;
const
   Black = 0;
   ....
```

Wieso die doppelte Deklaration der Konstanten *Black* (einmal global zum Programm, das zweite Mal innerhalb der Prozedur *xxx*) keine Fehlermeldung

des Compilers auslöst, haben wir bereits in Kapitel 2 »Die sieben Elemente der Programmierung« geklärt: Die Redeklaration innerhalb von *xxx* ist lokal zu dieser Prozedur. Die Units *Crt* und *Graph* enthalten aber ebenfalls Konstanten für die Bildschirmfarben – *Black* ist in diesem Programm also gleich vierfach deklariert. Daß der Compiler trotzdem keinen Fehler meldet, liegt daran, daß auch hier mit dem Prinzip lokaler Deklarationen gearbeitet wird. Im Prinzip behandelt Turbo Pascal unser Beispielprogramm in der folgenden Form:

```
unit System;      { ist immer der äußerste Block }
begin
  unit Crt;       { erste mit uses aufgenommene Unit }
  begin
    unit Graph;   { zweite mit uses aufgenommene Unit }
    begin
      ....        { weitere Units }
      program Mehrfach;
      begin
   ....
      end.        { des Programms }
      ...         { Ende weiterer Units }
    end.          { die Unit Graph }
  end.            { die Unit Crt }
end.              { die Unit System }
```

Die Unit *System* stellt immer den äußersten Block eines Programms dar, die nächsten Ebenen werden von aufgenommenen Units in der Reihenfolge gebildet, in der sie mit **uses** angegeben wurden. Das Programm selbst ist der innerste Block und kann Bezeichner beliebig redeklarieren. Dieses Prinzip gilt nicht nur für Konstanten, sondern auch für alle anderen Bezeichner. Ein weiteres Beispiel:

```
program GotoXYDemo;
uses
  Crt,        { enthält die Prozedur GotoXY }
  Turbo3;     { benutzt die Prozedur GotoXY }

procedure GotoXY (x,y: Integer);
  ...
```

Jeder Aufruf von *GotoXY* durch das Programm benutzt die Version von *GotoXY*, die auch innerhalb des Programms definiert ist. Aufrufe von *GotoXY* durch Routinen der Units *Turbo3* und *Crt* verwenden die Prozedur *GotoXY* der Unit *Crt*.

Hinweis: Die beiden Bezeichner *GotoXY* haben nichts miteinander zu tun – innerhalb des Programms ließe sich *GotoXY* auch problemlos als Funktion, Variable, Konstante oder Datentyp deklarieren.

Zugriffe auf mehrfach deklarierte Bezeichner

Die Redeklaration eines Bezeichners in einem untergeordneten Block hat normalerweise zur Folge, daß der ursprüngliche Bezeichner innerhalb dieses Blockes nicht mehr verfügbar ist. Anders gesagt: Wenn wir innerhalb des

Programms den Bezeichner *GotoXY* deklarieren, kann die Prozedur *GotoXY* der Unit *Crt* vom Programm aus nicht mehr aufgerufen werden. Turbo Pascal umgeht diese Beschränkung durch das Konzept der *qualifizierten Bezeichner* mit der folgenden Syntax:

```
Unitname.Bezeichner
```

Ein Bezeichner, dem ein Unitname und ein Punkt vorausgehen, bezieht sich auf diese Unit – unabhängig davon, ob innerhalb des momentanen Blockes eine Redeklaration stattgefunden hat. Einige Beispiele, die sich auf die vorangegangenen Abschnitte beziehen:

```
GotoXY(3,4);      { ruft die Prozedur GotoXY von GotoXYDemo auf }
Crt.GotoXY(3,4);  { ruft die Prozedur GotoXY der Unit Crt auf }
x := Black;       { verwendet die Konstante des Programms Mehrfach }
x := Graph.Black  { verwendet die Konstante der Unit Graph }
Graph.TextAttr := $41;  { weist der Variablen TextAttr der Unit
                          Graph den Wert $41 zu }
```

Dieses Konzept gilt für alle Bezeichner und Units. Die beiden folgenden Programme sind also identisch:

```
program Normal;
var s: string;
begin
  Write('Wie heißen Sie? ');
  Readln(s);
  Writeln('Hallo, ', s, chr(7));
end.
```

```
program Ueberqualifiziert;
var s: string;
begin
  System.Write('Wie heißen Sie? ');
  System.Readln(s);
  System.Writeln('Hallo, ', s, System.chr(7));
end.
```

Weitere Details zu qualifizierten Bezeichnern finden Sie in den Kapiteln 1 »Sprachelemente und Konstanten« und 2 »Blöcke, Lokalität und Gültigkeitsbereich« des *Programmierhandbuches*.

uses-Anweisungen im Implementations-Teil und gegenseitige Abhängigkeiten

Eine **uses**-Anweisung im Interface-Teil wie

```
uses MyUnit;
```

macht nicht nur – wie bereits des öfteren erwähnt – die globalen Bezeichner der Unit *MyUnit* innerhalb der gesamten Unit verfügbar, sondern legt damit auch

eine strenge Hierarchie fest: Wenn *UnitA* eine Unit *UnitB* mit **uses** aufführt, gibt es keine Möglichkeit, *UnitB* seinerseits mit **uses** in *UnitA* aufzunehmen.

Aus diesem Grund erlaubt Turbo Pascal zusätzliche **uses**-Anweisungen im Implementations-Teil: In dieser Weise aufgenommene Units sind sozusagen »lokal« und kommen sich auch dann nicht in die Quere, wenn sie sich gegenseitig voraussetzen. Das folgende Beispiel demonstriert eine solche Situation, die sich ansonsten nur mit zwei Versionen ein und derselben Routine lösen ließe:

```
program Im_Kreis;

uses Crt, Display;  { Die Unit Display enthält die Routine WriteXY }

const Msg = 'Zurück in bewohnte Gebiete';

begin
  ClrScr;
  WriteXY(1,1, 'Obere linke Ecke des Bildschirms');
  WriteXY(100,100, 'Ziemlich unwahrscheinlich...');
  WriteXY(81 - Length(Msg), 15, Msg);
end.
```

Im_Kreis verwendet eine Routine namens *WriteXY,* die in der Unit *Display* definiert ist und den Text ab der angegebenen Position auf den Bildschirm ausgibt:

```
unit Display;

interface
procedure WriteXY(X,Y: Integer; Msg: string);

implementation
uses
  Crt, Error;

procedure WriteXY(X,Y: Integer; Msg: string);
begin
  if (X > 0) and (X <= 81-Length(Msg)) and (Y > 0) and (Y <= 25) then
  begin
    GotoXY(X,Y);
    Write(Msg);
  end
   else PrintError('X/Y-Koordinaten außerhalb des Bildschirms!');
end;

end.
```

Die Routine *WriteXY* der Unit *Display* prüft, ob der als *Msg* übergebene String komplett auf den Bildschirm paßt – falls ja, wird er ausgegeben; falls nein, ruft sie eine Routine namens *PrintError* zur Ausgabe einer Fehlermeldung aus.

PrintError ist in einer Unit namens *Error* definiert, das von *Display* folgerichtig auch mit **uses** aufgeführt wurde:

```
Unit Error;

interface
procedure PrintError(ErrMsg: string);

implementation
uses
  Display;

procedure PrintError(ErrMsg: string);
begin
  WriteXY(1,25, 'Fehler: ' + ErrMsg);
end;

end.
```

Und hier würden die Probleme mit »öffentlichen« **uses**-Anweisungen anfangen: *PrintError* verwendet seinerseits die Routine *WriteXY* zur Ausgabe der Fehlermeldung.

Durch das »Verschieben« beider **uses**-Anweisungen in den Implementations-Teil kann Turbo Pascal die Interface-Teile komplett compilieren und damit den Aufruf von *WriteXY* in den Code von *PrintError* einsetzen, weil der Compiler zu diesem Zeitpunkt bereits »weiß«, wie die Deklaration von *WriteXY* aussieht.

Dieser Mechanismus gilt auch für mehrfache gegenseitige Abhängigkeiten (d.h. für Fälle, in denen sich drei oder mehr Units »kreisförmig« aufeinander beziehen). Als Definition formuliert:

Solange die Interface-Teile zweier Units nicht gegenseitig voneinander abhängig sind (z.B. aufgrund gemeinsamer Daten-Deklarationen), können sich Units gegenseitig voraussetzen: Die entsprechenden **uses**-Anweisungen werden im Implementations-Teil untergebracht; sie müssen dort unmittelbar auf das reservierte Wort **implementation** folgen.

Gemeinsame Datendeklarationen

Die Definition des vorangegangenen Abschnitts läßt gemeinsam benutzte Daten (und Routinen) bewußt aus. Tatsächlich müßten von beiden Units verwendete Variablen, Datentypen oder Routinen im Interface-Teil eines der beiden stehen – womit wir wieder beim alten Problem angelangt wären. Setzen wir beispielsweise einmal voraus, daß die Prozeduren *WriteXY* und *PrintError* unseres Programms *Im_Kreis* zusätzlich die Grenzen eines Ausgabefensters als Parameter erwarten, also:

```
procedure WriteXY(OutputWindow: WindRec;  { Fenstergrenzen }
                  X,Y: Integer;           { Position im Fenster }
                  Msg: string;)           { Text }

procedure PrintError(OutputWindow: WindRec;  { Fenstergrenzen }
                     X,Y: Integer;        { Position im Fenster }
                     Msg: string;)        { Text }
```

Da der Datentyp *WindRec* für beide Routinen zur Verfügung stehen muß, gibt es hier nur eine (aber recht einfache) Lösung: Definieren Sie eine dritte Unit, die nur die entsprechende Typ-Definition enthält:

```
unit WindData;
interface
type
  WindRec = record
              X1,Y1,X2,Y2: Integer;       { Fenstergrenzen }
              TColor, BkColor:  Integer;  { Textattribut }
              Active: Boolean;
            end;
implementation
end.
```

Weil diese Unit keine der beiden anderen voraussetzt, läßt sie sich ohne jedes Problem mit einer **uses**-Anweisung in die Interface-Teile von *Display* und *Error* aufnehmen. (Sowohl *Display* als auch *Error* arbeiten dann mit zwei **uses**-Anweisungen – einer im Interface- und einer im Implementations-Teil.)

Units und große Programme

Bis jetzt haben wir Units nur als Bibliotheken vorgestellt, d.h. als Sammlung von Routinen, die von mehreren Programmen benutzt werden. Units haben aber noch eine weitere wichtige Funktion; mit ihrer Hilfe kann man umfangreiche Programme in einzelne Module zerlegen.

Die Unterteilung eines großen Programms in Units hat eine ganze Reihe großer Vorteile:

- Units müssen (im Gegensatz zu Include-Dateien) nicht bei jeder Veränderung des Hauptprogramms separat compiliert werden – die Compilierungszeiten werden kürzer (soweit das überhaupt noch möglich ist).

- Jede Unit stellt ein eigenes Code-Segment zur Verfügung und kann bis zu 64 KByte Code umfassen – die Gesamtgröße des Programms ist nur noch durch den zur Verfügung stehenden Hauptspeicher begrenzt.

- Die modulare Strukturierung eines Programms und die damit verbundene Übersichtlichkeit ergibt sich bei der Verwendung von Units praktisch von selbst.

Ein sehr großes Programm wird man normalerweise in mehrere Abschnitte unterteilen. Zu einem Editor gehören beispielsweise eine Initialisierung, Funktionen für das Ausdrucken von Text, zum Einlesen und Schreiben von Dateien, dem Formatieren von Text usw. Außerdem empfiehlt sich eine »globale« (d.h. von allen anderen Programmteilen benutzte) Unit, die alle globalen Konstanten, Datentypen und Variablen definiert.

Das Grundgerüst eines Editors (auf den wir in Kapitel 8 »Der Editor von A bis Z« noch ausführlicher zurückkommen) könnte folgendermaßen aussehen:

```
program Editor;
uses
   Dos, Crt, Printer,    { Standard-Units von Turbo Pascal }
   EditGlobals,          {"globale" Unit}
```

```
    EditInit,              {Initialisierung}
    EditPrint,             {Ausdruck-Funktionen}
    EditRead,              {Dateilesen}
    EditWrite,             {Dateischreiben}
    EditFormat;            {Textformatierung}

    { Deklarationen, Funktionen und Prozeduren
      innerhalb von EDITOR }

begin
   {das Hauptprogramm von EDITOR}
end.
```

Die selbstgeschriebenen Units können sich hier entweder in eigenen .TPU-Dateien befinden oder mit TPUMOVER in die Datei TURBO.TPL kopiert worden sein. Beide Wege haben ihre Vorteile: Im ersten Fall übernimmt Turbo Pascal die Verwaltung (d.h. eine entsprechende Recompilierung nach Veränderungen der Quelltexte) automatisch, im zweiten Fall stehen die Units ohne Diskettenzugriffe zur Verfügung.

Overlays

Nachdem Turbo Pascal pro Unit bis zu 64 KByte Code erlaubt, steht dem Schreiben beliebig großer Programme eigentlich nichts mehr im Wege – außer der Architektur des IBM PC mit seinen 640 KByte Hauptspeicher, von denen bereits rund 100 KByte durch das Betriebssystem belegt sind.

Die endgültige Lösung dieses Problems trägt den Namen *Overlays* und soll hier erst einmal begrifflich definiert werden: Overlays sind Programmteile, die ein und denselben Bereich im Hauptspeicher des Computers belegen – aber zu verschiedenen Zeitpunkten. Erst wenn das Programm eine bestimmte Routine aus einem Overlay benötigt, lädt es den entsprechenden Code in den Speicher (und überschreibt dabei eventuell andere Programmstücke, die momentan nicht gebraucht werden).

Programme, die mit dieser Technik arbeiten, können einen (fast) beliebigen Umfang haben und sind in weiten Grenzen tatsächlich vom verfügbaren Hauptspeicher unabhängig.

Um dieses Konzept in die Praxis umzusetzen, sind zwei weitere Festlegungen notwendig, nämlich:

- was unter einem »Programmteil« zu verstehen ist, und
- wer oder was dafür sorgt, daß Routinen zu dem Zeitpunkt auch verfügbar sind, zu dem sie vom Programm benötigt werden.

Turbo Pascal arbeitet bei der Verwaltung auf der Unit-Ebene: Eine Unit ist der kleinste Programmteil, der sich als einzelnes Overlay deklarieren und zu einem gegebenen Zeitpunkt auf einmal in den gemeinsam benutzten Speicherbereich

laden läßt. Bei der Übersetzung eines Programms TEST.PAS, das Overlays verwendet, erzeugt der Compiler zwei Dateien:

TEST.EXE – das ausführbare (Haupt-)Programm;

TEST.OVR – in dieser Datei sind sämtliche Units von TEST gespeichert, die als Overlay deklariert wurden.

Die Verwaltung des Speicherbereichs, der gemeinsam von mehreren Overlays genutzt wird, übernehmen die Routinen der Unit *Overlay* – sie erledigen das Nachladen (und Entfernen) einzelner Units zum richtigen Zeitpunkt, Zugriffe auf die .OVR-Datei usw.

Tatsächlich geschieht diese Verwaltung praktisch unsichtbar für den Programmierer: Solange Sie einige (wenige!) Regeln einhalten, arbeitet *Overlay* sozusagen geräuschlos im Hintergrund.

Die technischen Hintergründe

Turbo Pascal reserviert bei der Arbeit mit Overlays einen zusätzlichen Pufferbereich zwischen dem oberen Ende des Stack und dem Anfang des Heap – in diesen Bereich werden als Overlay deklarierte Units geladen.

Dieser Puffer wird (wie die Bereiche für Stack und Heap) beim Start des Programms belegt und ist standardmäßig so klein wie möglich – er bietet exakt soviel Platz, wie die größte Overlay-Unit des Programms benötigt.

Hinweis: Overlay-Units enthalten nicht nur Code, sondern auch eine Tabelle mit Querverweisen, die ebenfalls im Overlay-Puffer untergebracht werden muß – eine entsprechend umfangreiche Unit vorausgesetzt, ist es deshalb möglich, daß das Programm für den Puffer mehr als 64 KByte des Hauptspeichers belegt.

Für ein Programm TEST.EXE, das mit Overlays arbeitet, sieht der Aufruf einer Routine *OvrX* durch einen Programmteil wie

```
procedure CallX;
  ...
for x:= 1 to 10 do
  OvrX(....);
  ...
```

im Prinzip so aus:

1. Zuerst wird geprüft, ob sich *OvrX* momentan im Speicher befindet. Wenn das nicht der Fall ist: Laden der entsprechenden Unit aus der Datei TEST.OVR (wobei eventuell Platz im Overlay-Puffer durch das Hinauswerfen anderer Units geschaffen werden muß);

2. Ausführung von *OvrX* in »Realzeit«: als Overlay deklarierte Routinen unterscheiden sich von normalen Programmteilen nur durch die Prüfung auf »im Speicher« – ansonsten enthalten sie denselben Code wie normale Routinen und werden auch genauso bearbeitet.

3. Danach muß geprüft werden, ob sich die Routine *CallX* momentan im

Speicher befindet – es ist sehr wohl möglich, daß *CallX* ebenfalls Bestandteil eines Overlays ist und durch den Ladevorgang von *OvrX* überschrieben wurde. Falls dem so ist, wird *CallX* erneut aus TEST.OVR in den Speicher geladen, bevor es mit der Programmausführung weitergeht.

Im Extremfall (nämlich dann, wenn sich *CallX* und *OvrX* in zwei verschiedenen Overlay-Units befinden), enthält die Ausführung der Schleife in *CallX* also insgesamt zwanzig Diskettenzugriffe, bei denen jedesmal eine komplette Unit in den Speicher gelesen wird.

Um solche Situationen zu entschärfen, haben wir in die Routinen von *Overlay* eine Menge »Eigenintelligenz« hineingepackt:

- Turbo Pascal unterstützt den Lotus/Intel/Microsoft-Standard (EMS) für Speichererweiterungen: Über den Aufruf einer entsprechenden Routine von *Overlay* läßt sich festlegen, daß die .OVR-Datei des Programms beim Start komplett in die EMS-Karte eingelesen wird. Das Nachladen von Overlay-Units geschieht danach im Form reiner Kopieraktionen (von der EMS-Karte in den Hauptspeicher) – was auch gegenüber der schnellsten Festplatte eine erhebliche Geschwindigkeitssteigerung bedeutet.

- Turbo Pascal lädt grundsätzlich soviele Units wie möglich in den Puffer hinein, dessen Größe sich über eine Routine von *Overlay* beliebig nach oben hin erweitern läßt.

- Wenn beim »Nachladen« eine andere Unit überschrieben werden muß, wirft die Overlay-Verwaltung jene Programmteile hinaus, die lange nicht mehr aktiv waren.

Die Verwaltung des Overlay-Puffers

Der Overlay-Puffer von Turbo Pascal läßt sich am besten als Ringpuffer beschreiben, mit einem Zeiger auf den Anfang und einen Zeiger auf das Ende des Puffers. Overlays werden immer am Anfang des Puffers geladen, so daß die bereits geladenen Overlays an das Ende des Puffers verschoben werden. Wenn der Puffer voll ist (also nicht mehr genug Platz zwischen Anfang und Ende frei ist), dann werden die Overlays am Ende des Puffers gelöscht, um Platz zu schaffen für die neuen Overlays.

Da der normale Speicher nicht als Ringpuffer angelegt ist, gehören zur tatsächlichen Implementierung des Overlay-Puffers noch einige weitere Zwischenschritte, damit sich der Puffer wie ein Ringpuffer verhält. Abbildung 3.1 veranschaulicht diesen Vorgang. Die Abbildung zeigt den Verbleib der Overlays, die in einen anfangs noch leeren Puffer geladen werden. Overlay *A* wird zuerst geladen, dann kommt *B*, dann *C* und schließlich *D*. Die schattiert unterlegten Bereiche zeigen den noch freien Pufferbereich an.

Wie Sie sehen, tun sich eine Reihe interessanter Dinge beim Übergang von Schritt 3 auf Schritt 4. Zuerst springt der Kopfzeiger in den unteren Bereich des Puffers, wodurch alle geladenen Overlays (und der Zeiger auf das Ende) verschoben werden. Diese Verschiebung ist deshalb erforderlich, damit der freie Bereich zwischen den Zeigern auf Anfang und Ende liegt. Als zweites mußte die Overlay-Verwaltung das Overlay *A* vom Ende des Puffers entfernen, um das

Overlay *D* laden zu können. Overlay *A* ist in diesem Fall das Overlay, das von allen vorhandenen Overlays am frühesten geladen wurde, und ist deshalb die erste Wahl, wenn ein Overlay entfernt werden muß. Die Overlay-Verwaltung macht dann am Ende des Puffers mit dem Auslagern weiter, wenn am Kopf andere Overlays geladen werden müssen. Jedesmal, wenn der Kopfzeiger wieder nach unten springt, muß die Verschiebung wiederholt werden.

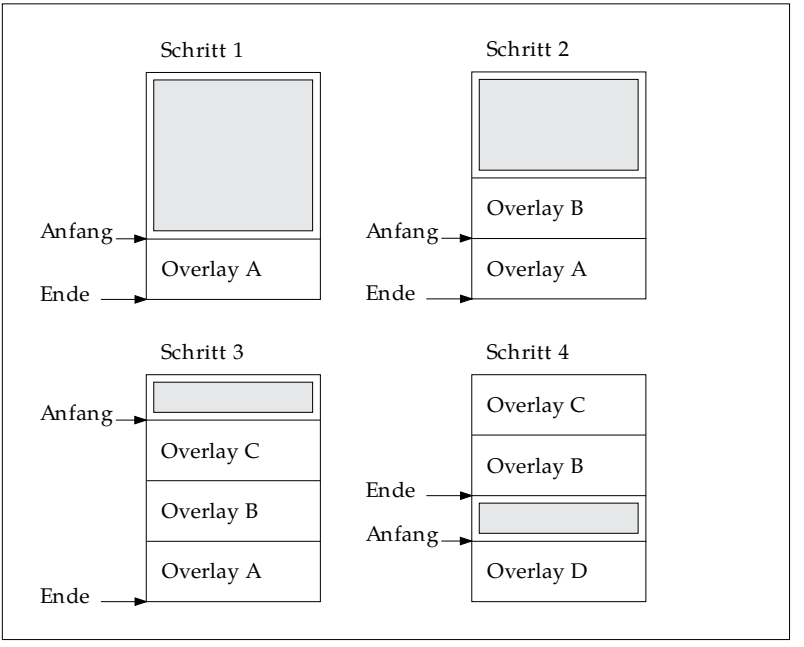


Abbildung 3.1 Laden und Freigeben von Overlays

Soweit die Art und Weise wie die Overlay-Verwaltung grundsätzlich funktioniert. Aber Turbo Pascal bietet auch eine zusätzliche Optimierung der Overlay-Verwaltung. Stellen Sie sich vor, daß Overlay *A* eine Reihe von sehr häufig benötigten Routinen enthält. Auch wenn diese Routinen fast immer benötigt werden, wird *A* hin und wieder aus dem Puffer entfernt und muß nachher aber sofort wieder geladen werden. Das Problem liegt darin, daß die Overlay-Verwaltung nichts über die Häufigkeit der Aufrufe der Routinen in *A* weiß.

Es ist lediglich bekannt, daß eine Routine in *A* aufgerufen wurde, und wenn *A* nicht im Speicher liegt, muß es erneut geladen werden. Eine Lösung für dieses Problem könnte so aussehen, daß man alle Aufrufe der Routinen in *A* registriert und dann nach jedem Aufruf einer Funktion in *A* dieses Overlay wieder an die Spitze des Overlay-Puffers verschiebt, damit klar ist, daß es das zuletzt verwendete Overlay ist. Das Abfangen der Aufrufe geht leider sehr zu Lasten der Ausführungsgeschwindigkeit und kann manchmal zu einer stärkeren Verlangsamung der Applikation führen, als dies durch die zusätzlichen Ladevorgänge des Overlays verursacht worden wäre.

Turbo Pascal bietet eine Kompromißlösung an, die die Leistungsfähigkeit praktisch nicht einschränkt, und trotzdem eine hohe Trefferquote bei der Suche nach häufig verwendeten Overlays, die nicht entfernt werden sollen, erhalten bleibt. Wenn ein Overlay nahe an das Ende des Overlay-Puffers rückt, erhält es eine »Bewährung«. Wenn während dieser Testphase ein Aufruf einer Routine in diesem Overlay erfolgt, erhält es einen »Aufschub« und wird nicht entfernt, wenn es ans Ende des Overlay-Puffers kommt. Stattdessen wird es einfach an den Anfang des Puffers gelegt und kann noch eine weitere Runde im Ringpuffer verbleiben. Wenn aber andererseits keine Aufrufe mehr in dieses Overlay während der Bewährungszeit erfolgen, wird das Overlay ausgelagert, falls es ans Ende des Overlay-Puffers kommt.

Im Endeffekt ergibt sich aus diesem Bewährung/Aufschub-Mechanismus, daß die häufiger verwendeten Overlays im Speicher bleiben, wobei nur ein *einziger* Aufruf des Overlays abgefangen wird, wenn sich das Overlay nämlich am Ende des Overlay-Puffers befindet.

Die beiden Verwaltungsroutinen, *OvrSetRetry* und *OvrGetRetry*, steuern den Mechanismus zwischen Aufschub und Bewährung. *OvrSetRetry* legt den Bereich des Overlay-Puffers fest, in dem sich die Routinen bewähren müssen, und *OvrGetRetry* liefert die aktuelle Einstellung. Wenn ein Overlay innerhalb der letzten Bytes von *OvrGetRetry* vor dem Ende des Overlay-Puffers liegt, wird es automatisch in den Bewährungsbereich geladen. Der gesamte freie Speicherplatz des Overlay-Puffers wird als Bewährungsbereich betrachtet.

... und die praktischen Voraussetzungen

Auf die Feinheiten der Programmierung kommen wir in einem eigenen Abschnitt noch zurück – hier sind die notwendigen Schritte erst einmal in Form einer Punkteliste wiedergegeben:

- Sämtliche Routinen des Programms müssen als **far** codiert sein (was sich am einfachsten erreichen läßt, indem man den Schalter {$F+} an den Anfang der jeweiligen Quelltexte setzt).

- Für die Übersetzung von Units muß der Compiler-Schalter *Overlays allowed* (= {$O+}) gesetzt werden – Turbo Pascal führt dann eine Reihe zusätzlicher Prüfungen bei der Compilierung aus. Wie der Name dieses Schalters impliziert, wird die entsprechende Unit damit nicht automatisch zum Overlay – der Compiler stellt hier nur sicher, daß sie als Overlay verwendet werden *kann*. Die Festlegung, *ob* die Unit auf normale Weise oder als Overlay eingebunden wird, erfolgt durch das Hauptprogramm. Anders gesagt: Mit {$O+} compilierte Units lassen sich ohne jede Einschränkung auch als normale Bestandteile eines Programms verwenden.

- Das Hauptprogramm muß die Unit *Overlay* vor allen anderen Units mit **uses** aufführen. Der Compiler-Befehl {$O <Unitname>} legt fest, welche Units als Overlays eingebunden werden:

```
program Test;
{$F+}      { FAR-Aufrufe für alle Routinen }

uses
  Overlay,     Crt, Dos,       Test1, Test2, Test3;
   { 1. Unit   / Bibliothek /  eigene Units }

{$O Test1}    { Test1 und Test3 als }
{$O Test3}    { Overlay, Test2 direkt im Programm }
  ....
```

- Die Initialisierung der Routinen von *Overlay* muß vor der Belegung dynamischer Variablen und (verständlicherweise) vor dem Aufruf der ersten Overlay-Routine erfolgen – am besten mit dem ersten Befehl des Hauptprogramms:

```
....
begin
  OvrInit('TEST.OVR');    { <- Name der Overlay-Datei }
```

- *OvrInit* bzw. die Overlay-Verwaltung erwarten die Overlay-Units in Form einer Datei auf der Diskette. Wenn Sie mit der integrierten Entwicklungsumgebung arbeiten, müssen Sie deshalb den Schalter *Compile/Destination* auf *Disk* setzen.

Weitere Hintergründe und einige »Vorsichts!«

Fast wäre man versucht zu sagen: »Das war's« – mehr als diese vier Punkte sind bei der Programmierung mit der Overlay-Verwaltung von Turbo Pascal eigentlich nicht zu beachten. Tatsächlich enthalten die folgenden Absätze keine weiteren Neuerungen, sondern nur zusätzliche Informationen:

- Der Platzbedarf des Overlay-Puffers wird vom Compiler automatisch als Minimalgröße des Heap (HeapMin) eingetragen bzw. zu einer bereits gesetzten Vorgabe hinzuaddiert – die Prüfung, ob für das Programm (inklusive des Puffers) genug Platz im Hauptspeicher zur Verfügung steht, findet deshalb bereits beim Programmstart statt. Eventuelle Fehlermeldungen in dieser Hinsicht werden also *nicht* durch die Overlay-Verwaltung, sondern von DOS (»Programm zu groß für den Hauptspeicher«) bzw. der integrierten Entwicklungsumgebung ausgegeben (»Not enough memory to run program«).

- Der Compiler kann selbständig erkennen, ob eine Unit mit {$O+} compiliert wurde oder nicht – im letzteren Fall reagiert er auf die Overlay-Deklaration dieser Unit (via {$O <Unitname>}) mit einer entsprechenden Fehlermeldung.

  Für den Schalter {$F+} existiert keine derartige Prüfung, weil sich lokal verwendete Overlay-Routinen auch als *near* codieren lassen (vgl. »Zur Programmierung« weiter hinten in diesem Kapitel).

  Vorsicht: Routinen, die als *near* codiert sind, enden beim Rücksprung von einem Overlay zum Hauptprogramm im Leeren, d.h. bringen das System mit ziemlicher Sicherheit zum Absturz.

- *Dos* ist die einzige Standard-Unit, die sich als Overlay deklarieren läßt. Alle anderen Bibliothek-Units (*System*, *Overlay*, *Crt*, *Graph*, *Turbo3* und *Graph3*) können nur in direkter Form in ein Programm aufgenommen werden.

- *OvrInit* (die Initialisierung der Overlay-Verwaltung) belegt Speicherplatz zwischen Stack und Heap, d.h. verschiebt den Heap-Beginn in Richtung höherer Speicheradressen – deshalb darf das Programm zu diesem Zeitpunkt noch keine dynamischen Variablen erzeugt haben.

Achtung: Das *Grafikpaket* von Turbo Pascal belegt bei seiner Initialisierung via *InitGraph* nicht nur Platz für den Grafiktreiber, sondern auch für interne Puffer auf dem Heap – der Aufruf von *InitGraph* muß deshalb auf jeden Fall nach dem Aufruf von *OvrInit* im Programm stehen.

Die Unit Overlay im Überblick

Overlay deklariert fünf Variablen und sieben Routinen, um deren konzeptuelle Zusammenhänge es in den folgenden Abschnitten geht – eine Beschreibung der technischen Details finden Sie in Kapitel 12 »Standard-Units« des *Programmierhandbuches* sowie in den entsprechenden Abschnitten von Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuches*.

OvrResult

```
var OvrResult: Integer;
```

OvrResult wird durch alle fünf Routinen von *Overlay* vor dem Rücksprung mit einem Statuscode gesetzt und stellt sozusagen das »Overlay-Analogon« zu den Standardfunktionen *IOResult* und *GraphResult* dar – mit zwei Unterschieden:

- diese Variable wird bei Zugriffen nicht automatisch auf den Wert 0 zurückgesetzt – »temporäre« Kopien vor einer Auswertung sind also nicht notwendig;
- *jeder* Aufruf einer Routine von *Overlay* setzt *OvrResult* erneut; eine Prüfung des »alten« Wertes (wie bei Ein-/Ausgabeoperationen im Modus {$I-}) findet nicht statt.

Die möglichen Werte für *OvrResult* sind in *Overlay* als Konstanten definiert:

Tabelle 3.1 Status-Konstanten von Overlay

Wert	Konstante	Beschreibung
0	ovrOk	Fehlerfreie Ausführung.
-1	ovrError	Allgemeine Fehlermeldung der Overlay-Verwaltung.
-2	ovrNotFound	Overlay-Datei (*.OVR) nicht gefunden.
-3	ovrNoMemory	Kein Platz zur Vergrößerung des Overlay-Puffers (via *OvrSetBuf*).
-4	ovrIOError	I/O-Fehler beim Zugriff auf die .OVR-Datei.
-5	ovrNoEMSDriver	EMS-Treiber nicht installiert (*OvrInitEMS*).
-6	ovrNoEMSMemory	Nicht genug Platz auf der EMS-Karte (*OvrInitEMS*).

OvrTrapCount

```
var OvrTrapCount : Word;
```

Jedesmal, wenn ein Aufruf einer Overlay-Routine von der Overlay-Verwaltung abgefangen wird, weil sich entweder das Overlay nicht im Speicher befindet, oder weil das Overlay im Bewährungsbereich liegt, wird die Variable *OvrTrapCount* um eins hochgezählt. Der Anfangswert von *OvrTrapCount* ist 0.

OvrLoadCount

```
var OvrLoadCount : Word;
```

Jedesmal, wenn ein Overlay geladen wird, wird die Variable *OvrLoadCount* um eins hochgezählt. Der Anfangswert von *OvrLoadCount* ist 0.

Indem Sie die Variablen *OvrTrapCount* und *OvrLoadCount* (zum Beispiel im Watch-Fenster des Debuggers) bei mehreren Aufrufen einer Applikation überwachen, können Sie die Auswirkung von verschieden großen Bewährungsbereichen (die mit *OvrSetRetry* festgelegt werden) verfolgen, um die optimale Größe für Ihr jeweiliges Anwendungsprogramm herauszufinden.

OvrFileMode

```
var OvrFileMode : Byte;
```

Die Variable *OvrFileMode* legt den Zugriffscode fest, der an MS-DOS beim Öffnen der Overlay-Datei übergeben wird. Standardmäßig enthält *OvrFileMode* den Wert 0, der einem reinen Lesezugriff entspricht. Indem man *OvrFileMode* vor dem Aufruf von *OvrInit* einen neuen Wert zuweist, kann man den Zugriffscode ändern, um zum Beispiel einen gemeinsamen Zugriff im Netzwerk zu ermöglichen. Weitere Informationen über die Werte für den Zugriffscode finden Sie im *DOS Programmer's Reference Manual*.

OvrReadBuf

```
type OvrReadFunc = function(OvrSeg : Word) : Integer;
var OvrReadBuf : OvrReadFunc;
```

Über die Prozedurvariable *OvrReadBuf* kann man in den Ladevorgang von Overlays eingreifen, um zum Beispiel eine Fehlerbehandlung zu installieren oder zu prüfen, ob ein austauschbares Speichermedium (Diskettenlaufwerk) funktionsbereit ist. Jedesmal, wenn die Overlay-Verwaltung ein Overlay lesen muß, wird die Funktion aufgerufen, deren Adresse in *OvrReadBuf* gespeichert ist. Wenn die Funktion den Wert 0 zurückgibt, geht die Overlay-Verwaltung davon aus, daß die Operation erfolgreich abgeschlossen wurde. Wenn das Funktionsergebnis ungleich 0 ist, wird der Laufzeitfehler 209 erzeugt. Der Parameter *OvrSeg* gibt an, welches Overlay geladen werden soll, aber wie Sie bald sehen werden, benötigen Sie diese Information überhaupt nicht.

Damit Sie Ihre eigene Funktion zum Lesen eines Overlays installieren können, müssen Sie zuerst den vorherigen Wert von *OvrReadBuf* in einer Variablen des Typs *OvrReadFunc* speichern und Ihre Funktion zum Lesen von Overlays an *OvrReadBuf* zuweisen. Innerhalb Ihrer Lesefunktion sollten Sie die gespeicherte

Leseroutine aufrufen, um das eigentliche Laden der Overlays durchzuführen. Irgendwelche Tests, die Sie durchführen wollen, wie zum Beispiel, ob ein austauschbares Speichermedium bereit ist, sollten vor dem Aufruf der vorher gespeicherten Lesefunktion vorgenommen werden. Alle Fehlerbehandlungen sollten erst nach dem Aufruf erfolgen.

Achtung: Ihre Overlay-Ladefunktion darf keine Routinen aufrufen, welche selbst in Overlays definiert sind. Ein solcher Aufruf führt zu einem Absturz des Systems.

Der Code, mit dem eine Funktion zum Lesen des Overlays installiert wird, sollte direkt hinter dem Aufruf von *OvrInit* stehen, wenn *OvrReadBuf* die Adresse der normalen Funktion zum Lesen vom Laufwerk enthält.

Wenn Sie zudem noch *OvrInitEMS* aufrufen, dann wird Ihre Funktion zum Lesen von Overlays von der Diskette in den EMS-Puffer verwendet, und wenn kein Fehler auftritt, wird die Adresse der Standardfunktion, um den EMS-Speicher zu lesen, gespeichert. Wenn Sie außerdem die EMS-Leseroutine ersetzen wollen, wiederholen Sie einfach den Installationsvorgang nach dem Aufruf von *OvrInitEMS*.

Die Standardfunktion zum Lesen von Laufwerken liefert im Erfolgsfall den Wert 0, ansonsten den Fehlercode von MS-DOS. In gleicher Weise gibt die Standard-EMS-Leseroutine den Wert 0 zurück, wenn alles klappt, und einen EMS-Fehlercode (mit Werten zwischen $80 und $FF), wenn die Operation scheitert. Weitere Informationen über die MS-DOS-Fehlercodes finden Sie im Abschnitt »Laufzeitfehler« des Kapitel 3 »Fehlermeldungen und -codes« des *Referenzhandbuches*. Die Details über die EMS-Fehlercodes stehen in der *Lotus/Intel/Microsoft Expanded Memory Specification*.

Der folgende Programmausschnitt zeigt, wie man eine Funktion zum Lesen von Overlays schreibt und installiert. Diese neue Funktion zum Lesen des Overlays wiederholt den Aufruf zum Laden des Overlays solange, bis keine Fehler mehr vorkommen. Alle Fehler werden an die Prozeduren *DOSError* und *EMSError* (die hier nicht aufgeführt sind) weitergegeben, so daß Sie den Fehler dem Anwender melden können. Beachten Sie, daß der Parameter *OvrSeg* an die gespeicherte Funktion zum Lesen des Overlays nur weitergegeben und von der neuen Overlay-Funktion niemals direkt verarbeitet wird.

```
uses Overlay;
var
  SpeichernOvrLesen : OvrReadFunc;
  EMSimEinsatz : Boolean;
{$F+}
function MeinOvrLesen(OvrSeg : Word) : Integer;
var
  E : Integer;
begin
  repeat
    E := SpeichernOvrLesen(OvrSeg);
    if E <> 0 then
      if EMSimEinsatz then
        EMSError(E) else DOSError(E);
```

```
  until E = 0;
  MeinOvrLesen := 0;
end;
{$F-}
begin
  OvrInit('MEINPORG.OVR');
  SpeichernOvrLesen := OvrReadBuf;    { Standardfehlerbehandlung
                                        für Disk-Zugriffe speichern }
  OvrReadBuf := MeinOvrLesen;         { Eigene Routine installieren }
  EMSimEinsatz := False;
  OvrInitEMS;
  if (OvrResult = OvrOK) then
  begin
    SpeichernOvrLesen := OvrReadBuf;  { EMS-Standardfehlerbehandlung
                                        speichern }
    OvrReadBuf := MeinOvrLesen;       { Eigene Routine installieren }
    EMSimEinsatz := True;
  end;
  {...}
end.
```

OvrInit

```
procedure OvrInit(Filename: string);
```

Initialisiert die Overlay-Verwaltung und öffnet die als *Filename* angegebene Datei (die die Overlay-Units des Programms enthalten muß). Wenn *Filename* keinen Suchweg und/oder Laufwerksbezeichner enthält, sucht *OvrInit* zuerst im momentan gesetzten Verzeichnis, dann in dem Verzeichnis, in dem die .EXE-Datei gespeichert ist (Voraussetzung: DOS-Version 3.x) und schließlich in sämtlichen Verzeichnissen, die über PATH angegeben sind.

Nach der Bestimmung, wieviel Platz die größte Overlay-Unit beansprucht, wird der Heap-Beginn entsprechend in Richtung höherer Speicheradressen verschoben und so Platz für den Overlay-Puffer geschaffen.

Mögliche Fehlercodes sind *ovrError* (unspezifisch) und *ovrNotFound* (.OVR-Datei nicht gefunden). Im Fehlerfall wird die Overlay-Verwaltung nicht installiert – nachfolgende Aufrufe von Overlay-Routinen haben den Abbruch des Programms mit dem Fehlercode 208 zur Folge.

Achtung: Alle anderen Routinen von *Overlay* (inklusive *ovrInitEMS*) setzen voraus, daß die Overlay-Verwaltung durch einen Aufruf von *OvrInit* initialisiert wurde.

OvrInitEMS

```
procedure OvrInitEMS;
```

Prüft, ob das System mit einer EMS-Karte ausgerüstet ist, die genug Platz für die Overlay-Datei bietet: Wenn ja, wird die .OVR-Datei vollständig in die EMS-Karte geladen, weitere Diskettenzugriffe durch die Overlay-Verwaltung sind dann nicht mehr notwendig.

Um Mißverständnissen vorzubeugen: Weil sich Programmteile nicht *in* der EMS-Karte ausführen lassen, bleibt das restliche Konzept (Overlay-Puffer, abwechselndes Laden von Overlay-Units usw.) dasselbe – *OvrInitEMS* ersetzt »lediglich« Diskettenzugriffe durch Kopieraktionen innerhalb des Speichers und beschleunigt so den Wechsel zwischen Overlays um ein Vielfaches.

Mögliche Fehlercodes sind *ovrError* (generischer Fehler), *ovrIOError* (Fehler beim Lesen der .OVR-Datei), *ovrNoEMSDriver* (kein EMS-Treiber installiert) und *ovrNoEMSMemory* (nicht genug Platz auf der EMS-Karte).

Hinweis: Ein erfolgloser Aufruf von *OvrInitEMS* hat keine negativen Seiteneffekte für das Programm – die Overlay-Verwaltung arbeitet in diesem Fall auf normale Weise weiter (d.h. mit Diskettenzugriffen).

OvrSetBuf

```
procedure OvrSetBuf(Size: LongInt);
```

Legt die Größe des Overlay-Puffers auf *Size* Bytes fest: Je größer der Puffer ist, desto mehr Overlay-Units lassen sich gleichzeitig im Speicher halten.

Der Heap-Beginn wird durch *OvrSetBuf* entsprechend der neuen Größe des Puffers nach unten bzw. oben verschoben. Woraus folgt: zum Zeitpunkt des Aufrufs von *OvrSetBuf* dürfen keine dynamischen Variablen belegt sein – falls doch, endet *OvrSetBuf* mit dem Statuscode *OvrError* und läßt den Puffer unverändert. (*New* und *GetMem* setzen ein spezielles Flag für die Overlay-Verwaltung – deshalb können Fehler dieser Art zuverlässig erkannt werden).

Die Minimalgröße des Puffers ist durch das umfangreichste Overlay des Programms vorgegeben, die Maximalgröße durch den momentanen Pufferumfang plus dem noch freien Platz auf dem Heap (der sich in diesem Fall durch *MemAvail* oder *MaxAvail* ermitteln läßt). Wenn die durch *Size* angegebene Größe außerhalb dieses Bereichs liegt, setzt *OvrSetBuf* den Status *ovrError* (»Puffer läßt sich nicht auf diese Größe verkleinern«) bzw. *ovrNoMemory*. Negative Seiteneffekte gibt es auch hier nicht – die Overlay-Verwaltung arbeitet einfach mit der alten Puffergröße weiter.

OvrGetBuf

```
function OvrGetBuf: LongInt;
```

Liefert die momentane Größe des Overlay-Puffers zurück. Nach der Initialisierung der Overlay-Verwaltung ist der Puffer so klein wie möglich, d.h. bietet exakt soviel Platz, wie die größte Overlay-Unit des Programms benötigt.

Hinweis: Da der Puffer nicht nur den Code der Overlay-Units, sondern auch zusätzliche Querverweise aufnehmen muß, kann sein Umfang auch für eine einzelne Unit die »magische« Grenze von 64 KByte übersteigen.

OvrClearBuf

```
procedure OvrClearBuf;
```

Entfernt sämtliche momentan geladenen Overlay-Units aus dem Puffer und erzwingt so ein »Nachladen« der entsprechenden Units durch die nächsten Aufrufe von Overlay-Routinen. *OvrClearBuf* ist ausschließlich für sehr spezielle

Anwendungen gedacht (wie z.B. das temporäre Belegen größerer Bereiche auf dem Heap oder eine explizite Neuordnung des Pufferinhalts durch gezielte Aufrufe von »Dummy-Routinen« einzelner Units) – normalerweise muß diese Funktion nie direkt aufgerufen werden.

Hinweise: Der Aufruf von *OvrClearBuf* durch eine Overlay-Routine hat ein Nachladen der entsprechenden Unit direkt nach dem Rücksprung von *OvrClearBuf* zur Folge (was folglich nicht allzuviel Sinn macht – es demonstriert hauptsächlich, daß die Overlay-Verwaltung von Turbo Pascal wirklich hieb- und stichfest ist). *OvrClearBuf* bezieht sich ausschließlich auf den Puffer – der Inhalt einer EMS-Karte wird von dieser Funktion nicht verändert.

OvrSetRetry

```
procedure OvrSetRetry(Groesse : Longint);
```

Die Prozedur *OvrSetRetry* legt die Größe des »Bewährungsbereichs« im Overlay-Puffer fest, wobei der Parameter *Groesse* angibt, wieviel Bytes *vor* dem Ende des Puffers der Bewährungsbereich beginnt. Wird nun ein Overlay in den so definierten Bereich verschoben, beginnt für dieses Overlay die Bewährungsphase. Aus Gründen der Kompatibilität zu früheren Versionen der Overlay-Verwaltung ist die Standardgröße des Bewährungsbereichs 0, wodurch der Bewährung/Aufschub-Mechanismus im Endeffekt ausgeschaltet wird. Nachfolgend ein Beispiel, wie *OvrSetRetry* eingesetzt werden kann:

```
OvrInit('MEINPROG.OVR');
OvrSetBuf(Puffergroesse);
OvrSetRetry(Puffergroesse div 3);
```

Es gibt (noch) keine empirische Formel, mit der man die optimale Größe des Bewährungsbereichs ermitteln könnte. Versuche haben jedoch gezeigt, daß Werte, die von einem Drittel bis zur Hälfte des Overlay-Puffers reichen, die besten Ergebnisse liefern.

OvrGetEntry

```
function OvrGetRetry;
```

Die Funktion *OvrGetRetry* ermittelt die aktuelle Größe des Bewährungsbereichs. Das bedeutet, es wird der Wert zurückgegeben, der in *OvrSetRetry* zuletzt gesetzt wurde.

Overlays in ».EXE«-Dateien

Mit Turbo Pascal können Sie die Overlays am Ende der .EXE-Datei Ihres Anwendungsprogramms und nicht nur in einer eigenen .OVR-Datei speichern. Um eine .OVR-Datei an das Ende einer ».EXE«-Datei anzuhängen, verwenden Sie den DOS-Befehl COPY mit der Kommandozeilenoption /B, wie z.B. in

```
copy /b meinprog.exe + meinprog.ovr
```

Sie müssen darauf aufpassen, daß die ».EXE«-Datei *ohne* die Informationen für Turbo Debugger compiliert wurde. Achten Sie also darauf, daß die Option *Options/Debugger/Debugging standalone* in der integrierten Entwicklungsumgebung nicht aktiviert ist und bei der Kommandozeilenversion des Compilers die Option /V nicht angegeben wird.

Um Overlays vom Ende einer ».EXE«-Datei und nicht aus einer eigenen ».OVR«-Datei zu lesen, brauchen Sie lediglich den Namen der ».EXE«-Datei im Aufruf von *OvrInit* anzugeben. Wenn Sie mit MS-DOS der Version 3.x arbeiten, können Sie die Standardfunktion *ParamStr* verwenden, um den Namen der ».EXE«-Datei zu ermitteln, wie zum Beispiel in

```
OvrInit(ParamStr(0));
```

Zur Programmierung

Die folgenden Seiten enthalten nicht nur einige wichtige Tips zur Arbeit mit Overlays – sie gehen auch noch einmal genauer (und mit einigen Beispielen) auf Dinge ein, die in den Abschnitten über die technischen Voraussetzungen nur als Punkteliste aufgeführt worden sind.

Der Compiler-Befehl {$O}

Wie bereits gesagt, läßt sich eine Unit nur dann als Overlay in ein Programm einbinden, wenn sie mit dem aktiven Schalter *O/C/Overlays allowed* (bzw. dem Compiler-Befehl {$O+} im Quelltext) compiliert wurde. Der Compiler nimmt in diesem Modus spezielle Prüfungen für String- und **set**-Konstanten vor, deren Sinn sich am besten anhand eines Beispiels verdeutlichen läßt: Gegeben sei eine Prozedur namens *WriteStr* in einer Unit *UnitA*, also

```
procedure WriteStr(s: string);
begin
  Writeln('Ein Text ', s);
end;
```

und eine Prozedur in einer anderen Unit *UnitB*, die *WriteStr* mit einer String-*Konstanten* aufruft:

```
...
WriteStr('...aus einer anderen Welt');
```

String-Konstanten werden normalerweise im Codesegment gespeichert (direkt vor der Routine, die sie definiert): in unserem Beispiel würden die Zeichen **...aus einer anderen Welt** also im Codesegment von *UnitB* landen. Der Aufruf von *WriteStr* übergibt folglich einen Zeiger auf dieses Codesegment – der ins Leere zeigt, wenn beide Units als Overlay deklariert sind und *UnitB* durch das Laden von *UnitA* überschrieben wird.

Im Modus {$O+} prüft der Compiler bei jedem Aufruf, ob er von einer Unit in eine andere Unit hineinführt: wenn ja, wird für die Übergabe strukturierter Konstanten eine interne Routine der Laufzeitbibliothek aufgerufen, die diese

Konstante auf den Stack kopiert – und anstelle eines Zeigers auf das Codesegment einen Zeiger auf diese »Stack-Kopie« übergibt.

Dieses Verfahren ist auch für »normale« Übergaben ohne jede Einschränkung anwendbar – *ob* eine mit {$O+} compilierte Unit später tatsächlich als Overlay deklariert oder direkt eingebunden wird, liegt in der freien Entscheidung des Programmierers. (Warum der Schalter *Overlays allowed* dann nicht grundsätzlich aktiviert ist? Weil jede Übergabe von String- und **set**-Konstanten durch den Aufruf dieser zusätzlichen Kopierroutine einige Mikrosekunden länger dauert und den Code jeweils um fünf Bytes verlängert).

Der Compiler-Befehl {$F+}

In den »technischen Voraussetzungen« ist ebenfalls aufgelistet, daß sämtliche Routinen von Overlay-Programmen als *far* codiert sein müssen. Diese Generalisierung ist nicht hundertprozentig korrekt: Tatsächlich muß nur sichergestellt werden, daß jede zum Zeitpunkt eines Overlay-Aufrufs *aktive* Routine mit dem Befehl RETF endet. Was in diesem Zusammenhang unter einer »aktiven« Routine zu verstehen ist, läßt sich am besten mit einem Beispiel zeigen:

Gegeben sei eine Prozedur namens *OvrX* in einer Overlay-Unit sowie zwei Routinen namens *Main_A* und *Main_B*, die im Hauptmodul des Programms (d.h. direkt in der .EXE-Datei) gespeichert sind. Nehmen wir weiterhin an, daß *Main_A* die Routine *Main_B* aufruft; *Main_B* enthält einen Aufruf von *OvrX*, also:

```
Main_A
       ---> Main_B
                   ---> OvrX
                        ....
                        ....
                  <---  RETF
            ....
 ....   <--- RETF
....
```

Zum Zeitpunkt des Aufrufs von *OvrX* sind hier die Routinen *Main_A* und *Main_B* aktiv (technisch: der Stack enthält sowohl eine Rücksprungadresse zu *Main_B* als auch eine Rücksprungadresse zu *Main_A*). Damit die Overlay-Verwaltung beim »Zurückrechnen« von Stack-Adressen nicht durcheinanderkommt, **müssen** diese beiden Routinen als *far* codiert sein.

Vorsicht: Wie üblich, interessiert sich der Compiler nicht für den Weg, den ein Programm nimmt – solange mit den Standardvorgaben gearbeitet wird, stellt er fest, daß sich *Main_A* und *Main_B* im selben Modul befinden, und codiert diese beiden Routinen als *near*. Zur Vermeidung derartiger Katastrophen sollte jeder Quelltext eines Overlay-Programms deshalb mit dem Schalter {$F+} beginnen.

Ähnliches gilt für Routinen, die sich im Implementations-Teil einer Unit befinden: sie müssen entweder explizit als *far* codiert sein – oder dürfen *nie* eine Routine aufrufen, die ihrerseits einen Overlay-Aufruf enthält. Theoretisch liegt ein Konstrukt wie

```
{$F+} Main_A
          ---> {$F+} Main_B
                              ------> {$F-} Main_B1
                                                ....
                             ...   <---------     RET
                             ...
                                    ------> {$F+} OvrX
                                                ....
                              ...   <----------   RETF
          <---------- RETF
```

– also eine explizit als *near* deklarierte Routine, die nur dann aufgerufen wird, wenn keine Overlays beteiligt sind – im Bereich des Möglichen; praktisch sollte man (außer in Ausnahmefällen) darauf verzichten. Außerdem: der Preis für die Codierung sämtlicher Routinen eines Programms als *far* ist schon fast lächerlich: pro Aufruf ein zusätzliches Wort auf dem Stack und ein einziges Byte an zusätzlichem Programmcode.

Einige ausgeführte Beispiele

Fangen wir mit der einfachsten aller Welten an – einem Programm OTEST.PAS, das zwei Overlays namens *T1_Ovr* und *T2_Ovr* definiert:

```
unit T1_Ovr;          { T1_OVR.PAS }
{$F+,O+}              { FAR-Codierung, "Overlays allowed" }
interface
uses t2_Ovr;
procedure Hello_T1;

implementation
procedure Hello_T1;
var x: Integer;

begin
  Writeln('Hier ist Overlay 1');
  for x:= 1 to 10 do
   Hello_T2;          { Routine des zweiten Overlays }
end;
end.   { Ende von T1_Ovr }
```

Wie zu sehen, haben wir hier bereits eine zusätzliche Hürde eingebaut: *T1_Ovr* setzt *T2_Ovr* voraus und ruft eine Routine dieser zweiten Unit wiederholt auf.

```
unit T2_Ovr;  { T2_OVR.PAS }
{$F+,O+}      { FAR-Codierung, "Overlays allowed" }
interface
procedure Hello_T2;

implementation
procedure Hello_T2;
begin
```

```
  Writeln('Und hier das zweite Overlay');
end;
end.          { Ende von T2_Ovr }
```

Die Unit *T2_Ovr* besteht in diesem einfachen Beispiel tatsächlich nur aus einer einzigen Routine.

```
program OverTest;    { OTEST.PAS }

{$F+}                { FAR-Aufrufe, "Overlays allowed"
                       ist hier nicht notwendig }
uses
   Overlay, T1_Ovr, T2_Ovr;

{$O T1_Ovr}
{$O T2_Ovr}          { T1_Ovr und T2_Ovr als Overlay-Units
                       deklariert }

begin
  OvrInit('OTEST.OVR');
  Hello_T1; {lädt T1_Ovr, führt Hello_T1 aus; Hello_T1 lädt T2_Ovr }
end.
```

Fehlerprüfungen beim Aufruf von *OvrInit* finden hier nicht statt – falls sich die Datei OTEST.OVR nicht finden oder lesen läßt, bricht das Programm beim Aufruf von *Hello_T1* mit dem Laufzeitfehler 208 (*Overlay manager not installed*) ab.

Bevor wir zu den entsprechenden Möglichkeiten und einer sinnvollen Reaktion des Programms kommen, erst einmal eine Verfeinerung (die sich für ein Programm der Komplexität von OTEST sicher nicht lohnt): Wenn Ihr Computer mit einer EMS-Karte ausgerüstet ist, dann läßt sich die Initialisierung der Overlay-Verwaltung mit einem »Umlagern« von OTEST.OVR in diese Karte erweitern:

```
begin
  OvrInit('OTEST.OVR');
  OvrInitEMS;
  Hello_T1;  {lädt T1_Ovr, führt Hello_T1 aus; Hello_T1 lädt T2_Ovr }
end.
```

Wenn bei der Initialisierung keine Fehler aufgetreten sind, prüft Turbo Pascal hier, ob eine EMS-Karte vorhanden ist – falls ja (und solange diese Karte genügend freien Speicherplatz bietet),wird die Datei OTEST.OVR vollständig in den EMS-Speicher gelesen. (Wie im vorletzten Abschnitt beschrieben, passiert auf Computern ohne EMS-Karte gar nichts – *OvrInitEMS* setzt den Status »keine EMS-Karte vorhanden« und die Overlay-Verwaltung arbeitet einfach auf normale Weise weiter – nämlich mit Zugriffen auf die Datei OTEST.OVR auf der Diskette.

Der Overlay-Puffer ist beim Start des Programms so bemessen, daß die größte Overlay-Unit des Programms (in unserem Beispiel: *T1_Ovr*) gerade hineinpaßt. Da *T1_Ovr* eine Routine von *T2_Ovr* aufruft, müssen diese beiden Units deshalb trotz aller Intelligenz der Overlay-Verwaltung ständig neu geladen werden. Die

Abhilfe liegt natürlich in einer direkten Vergrößerung des Puffers über einen Aufruf der Prozedur *OvrSetBuf*:

```
const OvrMaxSize = 200;   {für unser Beispiel mehr als ausreichend }
begin
  OvrInit('OTEST.OVR');
  OvrInitEMS;
  OvrSetBuf(OvrMaxSize);   { 200 Bytes Overlay-Puffergröße }
  Hello_T1;  {lädt T1_Ovr, führt Hello_T1 aus; Hello_T1 lädt T2_Ovr }
end.
```

Nun können *T1_Ovr* und *T2_Ovr* gleichzeitig im Speicher gehalten werden. (Einen Unterschied in der Ausführungsgeschwindigkeit von OTEST werden Sie aller Wahrscheinlichkeit nach nicht bemerken, weil beide Units zusammen weniger als 512 Bytes belegen und von DOS beim ersten Zugriff auf OTEST.OVR in einen Puffer im Hauptspeicher gelesen worden sind).

Noch einmal eine Erinnerung: *OvrSetBuf* verschiebt den Heap-Beginn und setzt deshalb voraus, daß zum Zeitpunkt des Aufrufs noch keine dynamischen Variablen erzeugt worden sind. Die Initialisierung des Grafikpakets via *InitGraph* muß also nach dem Festlegen der Puffergröße erfolgen.

Fehlerprüfung

Die folgende Version von OTEST berücksichtigt wirklich sämtliche Fehlermöglichkeiten bei der Initialisierung der Overlay-Verwaltung:

```
programOverTest;          { OTEST.PAS }

{$F+}                    { FAR-Aufrufe, "Overlays allowed"
                           ist hier nicht notwendig }
uses Overlay, T1_Ovr, T2_Ovr;

{$O T1_Ovr}
{$O T2_Ovr}              { T1_Ovr und T2_Ovr als Overlay-Units
                           deklariert }

constOvrMaxBuf = 2000;  { Größe des Overlay-Puffers }

var OvrName: string;

begin
  OvrName := 'OTEST.OVR');
  repeat
    OvrInit(OvrName);
    if OvrResult = OvrNotFound then

    begin
      Writeln('Overlay-Datei ',OvrName, ' nicht gefunden!');
      Write('Bitte den korrekten Dateinamen eingeben: ');
      Readln(OvrName);
    end;
  until ((OvrResult <> OvrNotFound) or OvrName = '');
  if OvrName = '' then
```

```
begin
  Writeln('Programm abgebrochen.');
  Halt;
end;
```

Wenn sich die Overlay-Datei nicht finden läßt, wird der Benutzer nach dem richtigen Dateinamen (und Suchweg) gefragt – solange, bis die Sache funktioniert (oder er keinen neuen Namen mehr eingibt, woraufhin das Programm abbricht).

```
if OvrResult <> OvrOk then
begin
  Writeln('Overlay-Fehler: ', OvrResult);
  Halt;
end;
```

Sollten sich bei der Initialisierung andere Fehler ergeben haben, wird das Programm an dieser Stelle beendet. (Fehler von *OvrInit* sind sozusagen endgültig – wer sie ignoriert, bekommt beim ersten Aufruf einer Overlay-Routine die Quittung in Form des Laufzeitfehlers 208 präsentiert).

```
OvrInitEMS;   { Haben wir eine EMS-Karte? }
if OvrResult <> OvrOk then
begin
  case OvrResult of
    OvrIOError:      Write('Fehler beim Lesen der Overlay-Datei');
    OvrNoEMSDriver:  Write('EMS-Treiber nicht installiert');
    OvrNoMemory:     Write('Nicht genug Platz auf der EMS-Karte');
    end;
  Write ('-- RETURN drücken'); Readln;
  end;
```

Für den Fall, daß die Prüfung bzw. das »Umlagern« via *OvrInitEMS* erfolglos bleibt, erhält der Benutzer eine entsprechende Meldung – aber ohne einen darauffolgenden Abbruch des Programms: Die Overlay-Verwaltung holt sich die notwendigen Daten im weiteren Verlauf von der Diskette.

```
  OvrSetBuf(OvrMaxSize);
  Hello_T1; { lädt T1_Ovr, führt Hello_T1 aus; Hello_T1 lädt T2_Ovr }
end.
```

Der Aufruf von *OvrSetBuf* (mit einem Wert, den Sie für Ihr eigenes Programm durch Experiment und Erfahrung ermitteln sollten), ist selbst in diesem Beispiel ohne nachfolgende Prüfung von *OvrResult* codiert. (Das einzig mögliche Ergebnis außer *OvrOk* wäre *OvrNoMemory* – zum einen läßt sich daran wohl nicht allzuviel ändern, zum anderen arbeitet die Overlay-Verwaltung in diesem Fall mit dem Original-Puffer störungsfrei weiter).

Möglichkeiten und Grenzen

Die folgenden Abschnitte befassen sich mit dem »Kleingedruckten« – mit Initialisierungs-Teilen (die theoretisch möglich sind), den nicht als Overlay deklarierbaren Standard-Units, interrupt-gesteuerten Routinen und ihren Problemen, der Fehlersuche über den integrierten Debugger und schließlich der Aufnahme von Maschinenroutinen. Dazu kommen einige Zusammenfassungen des Materials der vorangegangenen Seiten.

Initialisierungs-Teile in Overlay-Units
sind von der Codierung ohne weiteres möglich, weil sich Overlay-Units (abgesehen vom notwendigen Compiler-Schalter {$O+}) sonst in Struktur und Prozessorbefehlen nicht von normalen Units unterscheiden.

Die Probleme liegen in einem anderen Punkt – nämlich der Reihenfolge der Initialisierungen: Der Compiler setzt für jede Unit, die einen Initialisierungs-Teil enthält, einen Aufruf vor den eigentlichen Beginn des Hauptprogramms.

Da die Overlay-Verwaltung aber zuerst durch einen *Programmbefehl* (nämlich dem Aufruf von *OvrInit* zusammen mit einem Dateinamen) gestartet werden muß, findet das Programm ein recht schnelles Ende: Der Aufruf des ersten Overlay-Initialisierungs-Teils erzeugt den Laufzeitfehler 208, weil das entsprechende Overlay zu diesem Zeitpunkt mangels Verwaltung noch nicht verfügbar ist.

Es gibt allerdings ein logisch einwandfreies Hintertürchen: Wenn wir für unser Beispielprogramm OTEST davon ausgehen, daß *T1_Ovr* und *T2_Ovr* Initialisierungs-Teile enthalten (deren Aufrufe der Compiler – wie gesagt – unerbittlich vor dem ersten Befehl des Hauptprogramms stellt), dann kann man eine dritte (statische!) Unit erstellen, die ungefähr so aussieht:

```
unit T1T2_Init;
interface
uses Overlay;

implementation

begin
  OvrInit('OTEST.OVR');
  OvrInitEMS;
  OvrSetBuf(2000);
end.
```

Wenn diese Unit im Hauptprogramm *vor* allen Overlay-Units angegeben wird, also

```
program OverTest;       { OTEST.PAS }

{$F+}                  { FAR-Aufrufe, "Overlays allowed"
                         ist hier nicht notwendig }
uses Overlay, T1T2_Init, T1_Ovr, T2_Ovr;
```

```
{$O T1_Ovr}
{$O T2_Ovr}      { T1_Ovr und T2_Ovr als Overlay-Units deklariert }
```

dann führt Turbo Pascal seinen Initialisierungs-Teil auch vor den Aufrufen von *T1_Ovr* und *T2_Ovr* aus – und die Befehle im Initialisierungs-Teil von *T1T2_Init* starten die Overlay-Verwaltung.

Es gibt zwei Gründe, warum man diesen Weg nur in Ausnahmefällen beschreiten sollte:

- Der (hypothetische) Initialisierungs-Teil von *T1_Ovr* und *T2_Ovr* wird per Definition nur ein einziges Mal ausgeführt – trotzdem muß ihn das Programm ständig »mitschleppen«, entsprechend Platz im Overlay-Puffer reservieren usw. – was nicht nur Zeit, sondern vor allem auch unnötig Platz im Hauptspeicher kostet.

- Sämtliche Overlays, die Initialisierungs-Teile enthalten, werden beim Programmstart nacheinander geladen. Was bei OTEST wohl kaum auffallen dürfte, wird bei einem wirklich großen Programm zum echten Problem – wenn die Overlay-Verwaltung einige hundert Kilobyte nacheinander in den Speicher hinein- und wieder herausschaufeln muß, sieht sich der Benutzer recht bald unerfreulich langen Wartezeiten gegenüber.

Nach dieser langen Vorrede nun der kurze Lösungsvorschlag, der nicht nur Platz spart, sondern auch ohne zeitliche Nachteile ist:

Fassen Sie sämtliche notwendigen Initialisierungen in einer Overlay-Unit zusammen, die selbst keinen Initialisierungs-Teil hat. Diese Unit bzw. ihre Routinen sollten nach dem Start der Overlay-Verwaltung einmal (mit expliziten Befehlen) aufgerufen werden – und dann für die restliche Laufzeit des Programms überhaupt nicht mehr.

Was sich nicht als Overlay verpacken läßt...

haben wir größtenteils bereits besprochen:

- Units, die im Modus {$O-} compiliert wurden. Dazu gehören sämtliche Standard-Units außer *Dos* – also *System*, *Overlay*, *Crt*, *Graph*, *Turbo3* und *Graph3*.

- Units, die Interrupt-Behandlungsroutinen enthalten. Zum ersten hängt die absolute Adresse einer Unit davon ab, wo die Overlay-Verwaltung gerade Platz im Puffer gefunden hat; zum zweiten kann sie sich beliebig ändern; zum dritten: falls die Unit im Moment überhaupt nicht geladen ist, zeigt ein auf eine Interrupt-Routine gesetzter Vektor vollends ins Leere – mit entsprechenden Folgen. (Das ist auch der Grund, wieso sich *Crt* nicht als Overlay verwenden läßt: diese Unit definiert eine Behandlungsroutine für Ctrl Break).

Aufrufe über Zeiger, prozedurale Parameter und die Fehlersuche

Hier gibt es *keine* Einschränkungen irgendwelcher Art:

- Exit-Prozeduren, Gerätetreiber und selbstgeschriebene Routinen, die über Zeiger aufgerufen werden, lassen sich in Overlay-Units auf dieselbe Weise wie in einem statischen Programm verwenden;

- dasselbe gilt für die Übergabe von Overlay-Routinen als prozedurale Parameter, ihre Zuweisung an Variablen entsprechenden Typs usw.;
- und für die Fehlersuche mit dem integrierten Debugger: egal, ob es sich um das wahlfreie Setzen von Abbruchpunkten, die schrittweise Ausführung oder das Suchen von Routinen handelt – die Tatsache, daß sich der entsprechende Code in einer Overlay-Unit befindet, ist für den Programmierer sozusagen unsichtbar.

Der letzte Punkt bringt uns zu einem Tip, über den manch einer noch vor wenigen Monaten den Kopf geschüttelt hätte: Wenn sich ein Programm wegen Platzproblemen im Hauptspeicher nicht mehr mit den integrierten Debugger bearbeiten läßt, sollten Sie es mit einer Unterteilung in Overlays probieren – eventuell auch nur für die Dauer der Fehlersuche.

Assembler-Routinen und Overlays

Maschinensprache-Routinen, die als **external** bzw. über {$L <Dateiname>} in eine Overlay-Unit aufgenommen werden sollen, müssen dieselben Regeln wie in einem »normalen« Programm beachten.

Unabhängig davon, ob sich die Maschinenroutine in einem Overlay oder im Hauptmodul befindet, kommen dieselben Forderungen wie für Pascal-Routinen hinzu: Wenn das Maschinenprogramm eine Overlay-Routine aufruft (oder auch nur eine Routine, die ihrerseits einen Overlay-Aufruf enthält), dann **muß** sie als *far* codiert sein. Weitere Besonderheiten gibt es nicht, wie der folgende Ausschnitt demonstriert:

```
ExternProc   PROC FAR

             push bp       ; der übliche "Rahmencode"
             mov bp,sp     ; BP: Zeiger auf Parameter/lok. Variablen
             sub sp,xxxx   ; Platz für lokale Variablen
             ....
             ....
             call ProcX    ; Aufruf einer Overlay-Routine
             ....
             mov sp,bp     ; Freigabe der lokalen Variablen
             pop bp        ; BP-Original zurück
             ret yyyy      ; Freigabe der Parameter
ExternProc   ENDP
```

Wenn die Routine keine eigenen lokalen Variablen deklariert, können die beiden Befehle zum Setzen bzw. Wiederherstellen von SP wie üblich entfallen (vgl. Kapitel 15 »Der integrierte Assembler und Einbinden von Assembler-Code« des *Programmierhandbuches*) – das Setzen von BP ist dagegen auf jeden Fall notwendig, solange Overlay-Aufrufe direkt oder indirekt beteiligt sind.

Analog zu Pascal-Routinen, die weder direkte noch indirekte Aufrufe von Overlays enthalten, läßt sich auch ein Maschinenprogramm als **near** codieren – und auch hier sollte man defensiv arbeiten und auf diese Möglichkeit nur in Ausnahmefällen zurückgreifen.

Zwei weitere und letzte Forderungen gelten nur für Maschinenroutinen, die sich selbst in einer Overlay-Unit befinden:

- Variablen (und andere Manipulationen) im Codesegment sollten auf jeden Fall vermieden werden. Hier verbirgt sich ein Fallstrick, über den der Programmierer unter Umständen tagelang stolpern kann: Wenn die Overlay-Verwaltung den entsprechenden Platz im Overlay-Puffer zwischenzeitlich für eine andere Unit benötigt, wird beim nächsten Aufruf des Maschinenprogramms das *originale Codesegment* erneut von der Diskette bzw. EMS-Karte geladen – und damit auch wieder die »Originalwerte« eventueller Codesegment-Variablen.

- Entsprechendes gilt für (absolute) Zeiger auf Codesegmente von Overlays, die nur solange als gültig angenommen werden dürfen, wie die Programmausführung innerhalb ein und derselben Unit bleibt – in allen anderen Fällen ist es sehr wohl möglich, daß die Overlay-Verwaltung das entsprechende Segment verschoben oder ganz aus dem Speicher entfernt hat.

4. Programmierung für Fortgeschrittene

In Kapitel 2 »Die sieben Elemente der Programmierung« haben wir uns vorwiegend mit einfachen Datentypen wie Integern und einzelnen Zeichen beschäftigt – hier geht es nun um Strukturen höherer Ordnung, die sich aus diesen Grundtypen bilden lassen. Außerdem erläutern wir das Konzept dynamischer Variablen zusammen mit einfachen Listen und schließen die Einführung in die Grundelemente von Pascal damit ab.

Der zweite Teil dieses Kapitels baut auf diesen Grundlagen auf und ist nicht nur für Einsteiger interessant – wir demonstrieren einige Möglichkeiten der »PC-spezifischen« Programmierung über die Units *Dos* und *Crt* und gehen schließlich kurz auf das Thema »prozedurale Variablen« ein.

Datenstrukturen und Typen

Ein *Array* ist eine Sammlung gleichartiger Variablen mit einem gemeinsamen Bezeichner. Jede einzelne Variable eines Arrays wird als *Element* bezeichnet und über einen *Index* hinter dem Array-Namen angesprochen. Ein Beispiel:

```
var
  ZehnZeichen: array[1..10] of Char;
begin
  ZehnZeichen[1] := 'A';
  ZehnZeichen[5] := 'X';
  ...
```

Der Compiler stellt hier für die Variable *ZehnZeichen* 10 aufeinander folgende Speicherplätze zur Verfügung, in denen jeweils ein Wert des Typs *Char* gespeichert werden kann.

Wo liegt der Vorteil dieser Speicherform im Gegensatz zu zehn einzelnen Variablen, d.h. *Zeichen_1*, *Zeichen_2*...? Nun, hauptsächlich darin, daß der Zugriff auf die einzelnen Elemente als

```
Arrayname[Index-Ausdruck]
```

definiert ist: anstelle einer Konstanten wie 1 oder 5 läßt sich hier auch eine Variable oder das Ergebnis einer Berechnung einsetzen. Gehen wir gleich einen

Schritt weiter und schreiben wir ein Programm, das sich diese Tatsache für die Ausgabe von Wochentagen zunutze macht:

```
program Wochentag;
const
  Tage: array[1..7] of string[10] = ('Montag', 'Dienstag',
                                     'Mittwoch', 'Donnerstag',
                                     'Freitag', 'Samstag', 'Sonntag');
var
  x: Integer;
begin
  Write('Geben Sie eine Zahl von 1..7 ein: '); Readln(x);
  Writeln('Der Wochentag ', x, ' heißt: ', Tage[x]);
end.
```

Die einzelnen Elemente des Array *Tage* sind hier Strings mit einer Länge von maximal 10 Zeichen. Damit das Programm nicht allzu lang ausfällt, haben wir uns einen Vorgriff geleistet und eine *typisierte Konstante* verwendet (was einer Variablen entspricht, deren Wert zu Beginn des Programms gesetzt wird). Genauso hätten wir schreiben können:

```
var
  Tage: array[1..7] of string[10];
begin
  Tage[1] := 'Montag';
  Tage[2] := 'Dienstag';
  ....
```

Dieses Beispiel demonstriert noch einen weiteren Punkt: Die Elemente eines Array können ihrerseits wiederum einen beliebigen Typ haben – auch wenn sie auf spezielle Weise von Turbo Pascal behandelt werden, sind Strings im Prinzip nichts anderes als Arrays mit dem Elemententyp *Char*.

Ein schon fast klassisches Beispiel für Arrays, deren Elemente wieder aus Arrays bestehen, ist eine Schule mit 20 Schulklassen, die jeweils 30 Schüler haben. Jeder Schüler hat 12 Fächer, in denen er pro Jahr sechs Noten erhält, also

```
var
  Noten: array[1..6] of Integer;
  Schueler: array[1..12] of array[1..6] of Integer;
  Klasse: array[1..30] of array[1..12] of array[1..6] of Integer;
  Schule: array[1..20] of array[1..30]
          of array[1..12] of array[1..6] of Integer;
```

Besonders übersichtlich sieht das nicht gerade aus – genausowenig wie der Bezug auf die 3. Note des 7. Fachs des 23. Schülers der 2. Klasse, der als

```
Schule[2,23,7,3]
```

geschrieben werden muß. Versuchen wir das Ganze noch einmal und verwenden wir dabei eine der stärksten Fähigkeiten von Pascal – die Möglichkeit, eigene Datentypen zu vereinbaren:

```
type
  Noten = array[1..6] of Integer;
  Schueler = array[1..12] of Noten;
  Klasse = array[1..30] of Schueler;
  Schule = array[1..20] of Klasse;
var
  EineSchule: Schule;
```

Bevor wir das zweite Problem lösen, fügen wir gleich noch ein drittes hinzu: Nicht nur, daß es recht uneinsichtig ist, eine Note über den Bezeichner *Schule* anzusprechen – Schüler haben auch noch einen Vor- und Nachnamen, ein Alter, sind männlich oder weiblich... In vielen Programmiersprachen muß man sich damit behelfen, ähnliche Strukturen wie *Schule* (mit anderen Elemententypen) entsprechend oft zu deklarieren – zum x. Element des Noten-Array gehört dann das x. Element des Namen-Array, des Alter-Array usw.

Pascal erlaubt es, Variablen unterschiedlichen Typs in einem Verbund zusammenzufassen, der als *Record* bezeichnet wird. Ein Record mit Daten für einen unserer Schüler könnte so aussehen:

```
type
  SchuelerTyp = record
                  Vorname : string[20];
                  Nachname: string[30];
                  M_W     : Boolean;
                  Alter   : Integer;
                  Faecher : array[1..12] of Noten;  { siehe oben }
                end;
var
  Schueler: SchuelerTyp;
```

Die einzelnen Elemente dieses Records werden als *Felder* bezeichnet und über eine Kombination aus Variablennamen und dem *Feldnamen* angesprochen:

```
  Schueler.Vorname := 'Hänschen';
  Schueler.Alter := 15;
  Schueler.M_W := True;
   ...
```

Als letztes definieren wir einen *Aufzählungstyp* und bringen damit Ordnung in die Fächer und Noten:

```
type
  FaecherTyp = (Englisch, Deutsch, Mathematik, Erdkunde,
                Kunst, Sport, Musik, Chemie, Physik,
                Biologie, Ethik, Latein);
  Noten = array[1..6] of Integer;
  Fachnoten = array[FaecherTyp] of Noten;
var
  Fach : FaecherTyp;
  AlleFaecher: Fachnoten;
```

Nach dieser Vereinbarung können wir eine Variable des Typs *Fachnoten* über einen Ausdruck des Typs *FaecherTyp* ansprechen:

```
AlleFaecher[Englisch, 4] := 2;  { 4. Note im Fach Englisch }
Fach := Kunst;
AlleFaecher[Fach, 6] := 1;      { 6. Note im Fach Kunst }
```

Wenn wir nun das Feld *Faecher* im Record für unseren Schüler als *Fachnoten* deklarieren, also

```
type
  SchuelerTyp = record
                  Vorname : string[20];
                  Nachname: string[30];
                  M_W     : Boolean;
                  Alter   : Integer;
                  Faecher : Fachnoten;
                end;
var
  Schueler: SchuelerTyp;
```

dann geschieht der Zugriff auf die dritte Note im Fach Musik mit:

```
Schueler.Faecher[Musik, 3]
```

Zeiger und dynamische Variablen

Wir haben in Kapitel 2 »Die sieben Elemente der Programmierung« kurz angedeutet, daß Zeiger vor allem dann zum Einsatz kommen, wenn ein Programm mit »unbestimmten« Datenmengen arbeiten soll, d.h. die Anzahl der zu bearbeitenden Daten zum Zeitpunkt der Programmierung nicht bekannt ist.

Ein weiterer und vielleicht noch wichtigerer Grund für die Verwendung dynamischer Variablen liegt in der Architektur des Prozessors, die den Speicherplatz für (globale) statische Variablen auf 64 KByte begrenzt – die Verwaltung von mehr als 64 KByte Daten innerhalb des Hauptspeichers ist nur über Zeiger möglich.

»Unbestimmte« Datenmengen lassen sich auch in kleinem Maßstab recht einfach demonstrieren. Stellen Sie sich ein Programm vor, das eine Reihe von Eingaben erhalten, sie in Stringvariablen speichern und schließlich wieder ausgeben soll:

```
program Stringliste;   { Version 1, mit statischen Variablen }
var
  Puffer: array[1..20] of string;
  x,y : Integer;

begin
  x := 0;
```

```
  repeat                { bis zur Eingabe einer Leerzeile }
    x := x + 1;
    Write('Zeile ', x, ': ');
    Readln(Puffer[x]);
  until (Puffer[x] = '') or (x = 20);

  for y := 1 to x do    { Ausgabe der gespeicherten Werte }
    Writeln('Zeile ', y, ': ', Puffer[y]);

end.
```

Es ist offensichtlich, daß dieses Programm maximal 20 Eingaben speichern kann. Natürlich ließe sich das Array *Puffer* entsprechend vergrößern – aber bereits bei 256 Strings (mit jeweils 255 Zeichen) ist das Fassungsvermögen des Daten-Segments (64 KByte) erreicht. Die zweite Version von *Stringliste* umgeht zumindest die Beschränkung auf 256 Eingabezeilen:

```
programStringliste;        { Version 2, mit einem Zeiger-Array }
var
  Pufferzeiger: array[1..20] of ^string;
  x,y : Integer;
begin
  x := 0;
  repeat
    x := x + 1;
    New(Pufferzeiger[x]);   { erzeugt eine neue dynamische Variable }
    Write('Zeile ', x, ': ');
    Readln(Pufferzeiger[x]^);
  until (Pufferzeiger[x]^ = '') or (x = 20);

  for y := 1 to x do     { Ausgabe }
    Writeln('Zeile ', y, ': ', Pufferzeiger[y]^);

end.
```

Das Array trägt hier den Namen *Pufferzeiger* und speichert nicht die Stringvariablen selber, sondern Zeiger auf Strings. Der Aufruf von *New* belegt jeweils einen Bereich von 256 Bytes dynamisch und ordnet die Startadresse dieses Bereichs der angegebenen Zeigervariablen zu (also dem Element *x* des Array *Pufferzeiger*). Der Aufruf von *Readln* liest die Eingabe in den Speicherbereich ein, auf den *Pufferzeiger*[*x*] zeigt.

Da eine Zeigervariable lediglich 4 Bytes belegt, könnten wir das Array *Pufferzeiger* theoretisch auf rund 16300 Elemente vergrößern (und so über 4 Millionen Zeichen einlesen). Abgesehen davon, daß der Hauptspeicher maximal 640 KByte Platz bietet, bleibt aber auch hier das Problem, daß die Maximalzahl der einzulesenden Zeilen bei der Compilierung des Programms bekannt sein muß.

Die dritte und letzte Version von *Stringliste* ist ungleich komplizierter als ihre Vorgänger – sie verwendet die einfachste Form einer dynamisch verketteten Liste und kommt mit einer(!) statischen Variablen zur Speicherung aus. Die einzelnen Einträge dieser Liste sind folgendermaßen aufgebaut:

```
type
  PElement = ^TElement;
  TElement = record
               Daten : string;     { speichert eine Eingabezeile }
               Zeiger: PElement;   { zeigt auf den nächsten
                                     Record }
             end;
```

Zu Beginn des Programms wird der erste Record dynamisch erzeugt und seine Adresse in einer Zeigervariablen (*Wurzel*) festgehalten:

```
Wurzel ->   Daten
            Zeiger
```

Von der zweiten Eingabe an wird jeweils ein weiterer Record erzeugt und das Feld *Zeiger* des vorigen Records so gesetzt, daß es auf diesen neuen Record zeigt:

```
Wurzel ->   Daten
            Zeiger -> Daten
                      Zeiger
```

Dieser Prozeß läßt sich beliebig fortsetzen. Die (statisch deklarierte) Variable *Wurzel* wird nur bei der leeren Liste (Wurzel =**nil**) auf einen neuen Wert gesetzt und bleibt ansonsten unverändert. Wurzel zeigt damit immer auf den allerersten Record der Liste. Das Listing des Programms:

```
program Stringliste;       { Version 3, mit einer dynamischen Liste }

type
  PElement = ^TElement;    { Zeiger auf Records des Typs TElement }
  TElement = record
    Daten  : String;
    Zeiger : PElement;
  end;

var
  Wurzel,                        { enthält die Adresse des ersten
                                   Elements der Liste }
  Element       : PElement;      { enthält die Adresse des
                                   aktuellen Elements }
  Eingabe       : String;        { enthält die letzte Eingabe }
  x             : Integer;       { enthält die jeweilige Nummer
                                   der Eingabe }

procedure Anhaengen(var Erster, Letzter: PElement; s: String);
begin
  if Erster = nil then begin   { Bei der leeren Liste }
    New(Erster);               { wird der Listenkopf }
    Letzter := Erster;          { erzeugt. }
  end
```

```
  else begin
    New(Letzter^.Zeiger);       { Neuen Record an bestehende
                                  Liste hängen }
    Letzter := Letzter^.Zeiger;
  end;
  Letzter^.Daten := s;          { Felder des neuen Records setzen. }
  Letzter^.Zeiger:= nil;
end;

begin
  { --------------- Eingabe der Liste ---------- }
  x := 0;
  Wurzel:= nil;                 { Die Liste ist leer }
  repeat
    x := x + 1;                 { Einen Record anhängen }
    Write('Zeile ', x, ': ');
    Readln(Eingabe);
    if Eingabe  '' then Anhaengen(Wurzel, Element, Eingabe);
  until Eingabe = '';
  { ---------- Ausgabe der Liste ------------- }
  x := 0;
  Element := Wurzel;            { Element zeigt auf ersten Record }
  while Element <> nil do       { Wiederhole, solange das Listenende
                                  nicht erreicht ist }
    begin
      x := x + 1;
      Writeln('Zeile ', x, ': ', Element^.Daten);
      Element := Element^.Zeiger;  { Element zeigt auf nächsten
                                     Record }
    end;
end.                              { Ende von Stringliste }
```

Ein Durchlauf der Eingabeschleife läßt sich in fünf Schritte unterteilen:

1. Die Variable *Element* zeigt auf den »momentanen« (letzten Listen-) Record, den wir für diese Erläuterung mit *A* bezeichnen. Bei der ersten Eingabe ist Element undefiniert.

2. Mit dem Aufruf von *Readln* speichert die Variable *Eingabe* die String-Eingabe des Benutzers.

3. Gehen wir davon aus, daß der Benutzer etwas eingegeben (und nicht nur auf Enter gedrückt) hat: Der Aufruf *Anhaengen*(*Wurzel, Element, Eingabe*) belegt einen Speicherbereich für einen neuen Record *B* und weist sowohl Wurzel als auch *Element* die Startadresse dieses Bereiches zu. Damit enthält die Liste genau einen Eintrag, dessen Felder mit den entsprechenden Werten belegt werden.

4. *Element* zeigt nun auf die Adresse des neu erzeugten Records *B* und steht für den nächsten Schleifendurchlauf als letztes Element der Liste zur Verfügung, dem in der Prozedur *Anhaengen* jeweils ein weiteres Element angehängt wird.

5. Wird jedoch keine Eingabe gemacht (der Anwender hat nur Enter gedrückt), dann wird kein weiterer Speicherbereich belegt, und die Schleife bricht ab.

Der Ausgabeteil des Programms zeigt, wie einfach das Auslesen einer bestehenden Liste sein kann. Vor dem Start der Schleife wird *Element* auf den ersten Record gesetzt, dessen Adresse wir zu Beginn des Programms in der Variablen *Wurzel* festgehalten haben. Die Schleife selbst enthält nur zwei logische Schritte:

1. Ausgabe des Feldes *Daten* des Records, auf den *Element* zeigt;

2. Zuweisung der Adresse des nächsten Records in der Liste an *Element* und Wiederholung des Kreislaufs bis zum Erreichen des Listenendes.

(Die Konstante **nil** steht für »not in list« und ist ein spezieller Zeigerwert, der »nirgendwohin« zeigt. Wir haben das Feld *Zeiger* des letzten Records vor dem Abbruch der Eingabeschleife mit diesem Wert gesetzt und kennzeichnen so das Ende unserer Liste). Selbst das dritte Programm ist hinsichtlich des Speicherverbrauchs und der Ausführungsgeschwindigkeit nicht optimal. Folgende Änderungen führen (unter anderem) zu weiterer Effizienz-Steigerung:

Wird der Code des Hauptprogramms in einer Prozedur untergebracht, können die Hilfsvariablen *Element*, *Eingabe* und *x* lokal definiert sein und belegen nur temporär Speicherplatz.

Der Ausdruck (*Length*(*Eingabe*) = *0*) ist schneller und kürzer als der gleichwertige Ausdruck (*Eingabe* = ''').

Ist das Record-Feld Daten vom Typ ^*String*, kann dessen Speicherbedarf auf die tatsächliche String-Länge (plus 1 Byte) reduziert werden.

Dateioperationen

Für ein Pascal-Programm macht es keinen Unterschied, ob ein Zeichen zum Bildschirm oder in eine Datei auf die Diskette geschrieben wird – sowohl *Read* und *Readln* als auch *Write* und *Writeln* arbeiten grundsätzlich mit einer *Datei-Variablen* als erstem Parameter. Die vollständige Deklaration dieser Prozeduren ist:

```
Read(var Datei_Var; var Element; var Element...);
Readln(var Datei_Var; var Element; var Element...);
Write(var Datei_Var; Element, Element...)
Writeln(var Datei_Var; Element, Element...)
```

Daß wir den ersten Parameter bis jetzt folgenlos unterschlagen konnten, hat drei Gründe:

- Turbo Pascal stellt jedem Programm über die Unit *System* automatisch die globalen Datei-Variablen *Input* und *Output* zur Verfügung.

- Der Initialisierungsteil von *System* ordnet der Variablen *Input* die Tastatur zu und öffnet sie für Leseaktionen; *Output* bekommt den Bildschirm zugeordnet

und wird für Schreibaktionen geöffnet. Da der Initialisierungsteil von *System* beim Start eines Programms als erste Routine ausgeführt wird, sind diese Aktionen für alle anderen Units und Programmteile unsichtbar.

■ Wenn bei Aufrufen von *Read* und *Write* keine Datei-Variable angegeben ist, dann setzt der Compiler automatisch *Input* bzw. *Output* an der entsprechenden Stelle ein. Die beiden folgenden Aufrufe sind jeweils identisch:

```
Writeln(Output, 'Hallo!');
Writeln('Hallo!');

Readln(Input, x);
Readln(x);
```

Details der internen Struktur von Datei-Variablen finden Sie in Kapitel 12 »Standard-Units« des *Programmierhandbuches* bei der Besprechung der Unit *Dos*, die Kapitel 10 »Ein- und Ausgabe« des *Programmierhandbuches* und die Abschnitte über *Read* und *Write* in Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuches* erläutern die formalen und programmiertechnischen Einzelheiten.

Wir beschränken uns hier auf die Feststellung, daß Turbo Pascal drei Dateitypen unterscheidet, und gehen in den nächsten Abschnitten kurz auf die jeweiligen Charakteristika ein.

Hinweis: Der Dateityp bestimmt die Art und Weise, wie Turbo Pascal eine Datei behandelt, legt also eine *Interpretationsweise* von Daten fest. Physikalisch wird eine Datei auf der Diskette immer in derselben Form gespeichert – nämlich als Folge einzelner Bytes. Es ist deshalb möglich, eine als *Text* geschriebene Datei zu einem späteren Zeitpunkt beispielsweise als **file of** *Byte* wieder zu lesen.

Textdateien

Eine Textdatei wird als Folge von »Zeilen« betrachtet, wobei eine Zeile aus einer beliebigen Anzahl von Zeichen besteht, auf die ein Zeilenvorschub folgt. Da die Komponenten (=Zeilen) keine einheitliche Länge haben, sind berechnete Positionierungen nicht definiert: um beispielsweise die 133. Komponente einer Textdatei zu erreichen, müssen die davorliegenden 132 Zeilen der Reihe nach gelesen werden.

Input und *Output* sind als Textdatei-Variablen deklariert. Lese- und Schreiboperationen über *Read* und *Write* führen eine *automatische Interpretation* aus, deren Art vom Typ des übergebenen Ausdrucks abhängt:

```
Readln(Int_Var);    { liest eine ASCII-Zeichenfolge und berechnet
                      aus den angegebenen Ziffern einen 16-Bit-Wert }
Writeln(Int_Var);   { konvertiert einen 16-Bit-Wert in eine
                      Folge von ASCII-Zeichen }
```

Dateien des Typs *Text* können für Lese- *oder* Schreiboperationen geöffnet sein, nicht aber für beide Zugriffsarten gleichzeitig.

Hinweis: Die Prozeduren *Readln* und *Writeln* sind (im Gegensatz zu *Read* und *Write*) nur auf Textdateien anwendbar.

Das folgende Beispielprogramm öffnet eine Datei für Leseoperationen und gibt ihren Inhalt zeilenweise auf den Bildschirm aus:

```
program TextDump;
var
  f : Text;   { Deklaration einer Textdatei-Variablen }
  Name, Zeile : string;
begin
  Write('Name der Datei: ');
  Readln(Name);
  Assign(f, Name);       { ordnet der Datei-Variablen den
                           angegebenen Namen zu }
  Reset(f);              { öffnet die Datei für Leseaktionen }
  while not Eof(f) do    { solange das Ende dieser Datei }
  begin                  { nicht erreicht ist:           }
    Readln(f, Zeile);    {    eine Zeile lesen           }
    Writeln(Zeile);      {    Ausgabe auf den Bildschirm }
  end;
  Close(f);              { schließt die Datei }
end.
```

Typisierte Dateien

Dateien dieser Art bestehen aus einer Folge von Komponenten des gleichen Typs. Die Datei-Variable wird dementsprechend als

```
var
  file of Komponententyp;
```

deklariert, wobei *Komponententyp* für eine beliebige Datenstruktur steht. Da jede Komponente einer typisierten Datei denselben Platzbedarf hat, sind hier berechnete Positionierungen möglich, wie sie die folgenden Programmfragmente demonstrieren:

```
program  Verzeichnis;
type
  Plattentyp = record           { speichert eine Schallplatte }
      Titel   : string[20];
      Band    : string[20];
      Minuten : Integer;
    end;

var
  f      : file of Plattentyp;   { eine Datei, die aus Komponenten }
  Platte : Plattentyp;           { des Typs Plattentyp besteht }
  Nummer : Integer;
begin
  Assign(f, 'KLASSIK');         { Zuordnung der Datei zur Datei-
```

```
                                Variablen }
  Reset(f);                   { setzt voraus, daß die Datei
                              { bereits existiert }
    Write('Nummer der Schallplatte (0 = Ende): ');
    Readln(Nummer);
    if Nummer <> 0 then begin
      Seek(f, Nummer-1);      { setzt die Position in der Datei auf
                                die entsprechende Komponente }
      Read(f, Platte);        { liest diese Komponente }
      with Platte do begin
        Writeln('Titel: ', Titel);
        Writeln('Band:  ', Band);
        Writeln('Länge: ', Minuten);
      end;
      ....
    end;
    ....
```

Bei typisierten Dateien haben alle Operationen mit dem Dateiinhalt den Komponententyp zur Grundlage – auch der Dateiumfang wird nicht in Bytes, sondern als Anzahl der gespeicherten Komponenten angegeben. Im Gegensatz zum Typ *Text* sind bei typisierten Dateien nach einer Eröffnung mit *Reset* sowohl Lese- als auch Schreibzugriffe erlaubt.

Hinweise: Readln und *Writeln* sind auf typisierte Dateien nicht anwendbar. Der klassische Mechanismus (*Get* und *Put* zusammen mit einem Dateizeiger) ist in Turbo Pascal durch »intelligente« Versionen von *Read* und *Write* ersetzt.

Untypisierte Dateien

Bei diesem Dateityp macht Turbo Pascal überhaupt keine Annahmen über die Art und Organisation der gespeicherten Daten. Das Löschen oder Umbenennen ist wie bei den anderen Dateitypen möglich, Lese- und Schreiboperationen sind nur blockweise über die Prozeduren *BlockRead* und *BlockWrite* definiert:

```
program Typenlos;
{ ein einfaches Kopierprogramm OHNE jede Fehlerprüfung }

var
  Quelle, Ziel: file;
  R_gelesen, R_geschrieben: Word;
  Puffer: array[1..2048] of Byte;

begin
  { Name der Quell-Datei: 1. Kommandozeilen-Parameter }
  { Name der Ziel-Datei:  2. Kommandozeilen-Parameter }

  { Zuordnung und Öffnen der Quell-Datei }
```

```
  Assign(Quelle, ParamStr(1));
  Reset(Quelle, 1);                { Recordlänge: 1 Byte }

  { Zuordnung und Erzeugen der Zieldatei }
  Assign(Ziel, ParamStr(2)); Rewrite(Ziel, 1);

  Writeln(FileSize(Quelle), ' Bytes werden kopiert...');
  repeat
   BlockRead(Quelle, Puffer, SizeOf(Puffer), R_gelesen);
   BlockWrite(Ziel, Puffer, R_gelesen, R_geschrieben);
  until (R_gelesen = 0) or (R_gelesen <> R_geschrieben);
  Close(Quelle); Close(Ziel);
end.
```

PC-spezifische Programmierung

Turbo Pascal definiert über die Units *Crt*, *Dos* und *Graph* eine Vielzahl von Routinen, die weit über den Standard-Sprachumfang von Pascal hinausgehen. Wir werden uns in den folgenden Abschnitten mit einzelnen Aspekten und Möglichkeiten dieser Erweiterungen beschäftigen. (Eine detaillierte Besprechung aller Prozeduren und Funktionen finden Sie in Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuches*, einen tabellarischen Überblick in Kapitel 12 »Standard-Units« des *Programmierhandbuches*).

Aufrufe von MS-DOS

Die Standard-Unit *Dos* stellt die Schnittstelle zum Betriebssystems des IBM PC dar – hier sind die meisten Aufrufe von DOS und BIOS als Pascal-Routinen zusammen mit entsprechenden Datentypen deklariert. *Dos* setzt keine anderen Units voraus und kann deshalb einfach mit der Anweisung

```
uses Dos;
```

in ein Programm aufgenommen werden. Das folgende Beispiel ist ein Äquivalent des DOS-Befehls `dir`. Es baut hauptsächlich auf Routinen, Konstanten und Datentypen auf, die in der Unit *Dos* deklariert sind.

```
program Dateiliste;
uses Dos;                        { Aufnahme der Unit Dos }
var
  Suchweg: string;
  SRec : SearchRec;              { ein in der Unit Dos definierter
                                   Datentyp }
  KeinEintrag: Boolean;          { Flag für "keine Einträge gefunden" }

<  ... weitere Routinen des Programms >
```

```
begin
  repeat
    KeinEintrag := True;      { Annahme: keine Einträge gefunden }
    Write('Geben Sie den Namen/Suchweg ein (nur <RETURN> = Ende): ');
    Readln(Suchweg);
    if Suchweg <> '' then begin
      FindFirst(Suchweg, AnyFile, SRec);  { Suche nach 1.Vorkommen }
      while DosError = 0 do begin
        SRecAusgabe(SRec);    { Solange weitere Einträge vorhanden }
        FindNext(SRec);       { sind: Ausgabe / nächster Eintrag }
        KeinEintrag := False; { zumindest ein Eintrag ist vorhanden }
      end;
      if KeinEintrag then Writeln('<Keine Dateieinträge>');
      Writeln;
    end;
  until Suchweg = '';
end.
```

Hinweis: Die **uses**-Anweisung muß direkt hinter der **program**-Deklaration stehen.

Wir verwenden in diesem Programmfragment fünf Bezeichner, die in der Unit *Dos* deklariert sind: den Datentyp *SearchRec*, die Konstante *AnyFile* (=»jede Datei«) sowie die Routinen *FindFirst* und *FindNext*, die ein Verzeichnis nach Dateieinträgen absuchen. Die Variable *DosError* wird durch die Aufrufe von *FindFirst* und *FindNext* mit einem Ergebniscode gesetzt.

Die Prozedur *SRecAusgabe* läßt bereits aufgrund ihres Namens vermuten, daß sie nicht in der Unit *Dos*, sondern innerhalb des Programms deklariert wird. Sie gehört an die Stelle, die wir mit `< weitere Routinen >` gekennzeichnet haben:

```
procedure SRecAusgabe(D_Eintrag: SearchRec);
var
  DT: DateTime;

begin
  with D_Eintrag do begin
     Namenausgabe(Name);          { Ausgabe des Dateinamens }
     if (Attr and Directory) <> 0
       then Write ('<DIR>')       { <- Dateieintrag ist Directory }
       else begin
         Write(Size:10, '   ');   { ansonsten: Dateigröße }
         UnpackTime(Time,DT);
         Datumausgabe(DT);        { und Ausgabe von Datum/Uhrzeit }
       end;
     Writeln;
   end  { with D_Eintrag }
end;    { von SRecAusgabe }
```

Auch hier haben wir wieder starke Anleihen gemacht: Die Datentypen *DateTime* und *SearchRec*, die Prozedur *UnpackTime* und die Konstante *Directory*

sind sämtlich in der Unit *Dos* deklariert. *Name*, *Attr* (Attribut), *Size* (Dateiumfang) und *Time* (Datum/Uhrzeit) sind Feldnamen des Record-Typs *SearchRec*.

Bei *Namenausgabe* und *Datumausgabe* handelt es sich wieder um Prozeduren, die im Programm deklariert sind und dort vor *SRecAusgabe* stehen müssen:

```
procedure ZweiZiffern(I: Integer);  { wird von Datumausgabe benutzt }
begin                  { stellt Werten < 10 eine führende 0 voran }
  if I >= 10
    then Write(I:2)
    else Write('0', I:1);
end;

procedure Datumausgabe(DT: DateTime);
begin
  with DT do begin
    ZweiZiffern(Day); Write('.');           { Tag }
    Write(Month:2,'.');                     { Monat }
    ZweiZiffern(Year mod 100); Write('  '); { Jahr }
    ZweiZiffern(Hour); Write(':');          { Stunden }
    ZweiZiffern(Min);                       { Minuten }
  end;                                      { with DT }
end;
procedure Namenausgabe(Name: string);
var
  PunktPos: Integer;         { sucht den '.' im Dateinamen }
  Suffix: string[3];         { und trennt das Suffix ab }
begin
  PunktPos := Pos('.', Name);
  if PunktPos <> 0 then begin
    Suffix := Copy(Name, PunktPos+1, Length(Name)-PunktPos);
    Delete(Name, PunktPos, 1+Length(Name)-PunktPos);
   end
    else Suffix := '';
  Write(Name, ' ':(10-Length(Name)), Suffix, ' ':(5-Length(Suffix)));
end;
```

Dieses Programm ist nur ein kleines Beispiel für die Möglichkeiten, welche die Unit *Dos* bietet. Neben weiteren Routinen zur Bearbeitung von Verzeichnissen und Dateieinträgen sowie dem Abfragen und Setzen der Systemzeit finden Sie hier auch zwei Prozeduren, mit denen Aufrufe beliebiger DOS- und BIOS-Funktionen möglich sind: *Intr* und *MsDos*. Beide sind in Kapitel 14 »Interne Details« des *Programmierhandbuches* ausführlich besprochen.

Direkte Steuerung des Bildschirms

Die Unit *Crt* bildet eine Schnittstelle zur Bildschirmausgabe des Computers. Hier finden sich nicht nur alle Routinen, Konstanten und Datentypen, die in

der Version 3.0 von Turbo Pascal zum Standard-Sprachumfang gehören, sondern auch noch einige Neuigkeiten wie direkte Zugriffe auf den Bildspeicher (im Gegensatz zu BIOS-Routinen) und verschiedene boolesche Variablen wie *CheckBreak* (Abbruchmöglichkeit mit Ctrl Break), *CheckSnow* (verhindert »Schneegestöber« bei älteren CGA-Karten) und *CheckEof* (unterdrückt Ctrl Z -Eingaben). Details finden Sie (wie zu den Datentypen und Variablen von *Dos*) in Kapitel 12 »Standard-Units« des *Programmierhandbuches*.

Das folgende Beispielprogramm ist ein sehr einfacher »Editor«, der die Eingabe von Text in einem eigenen Fenster erlaubt und einige grundlegende Tastenbefehle zur Bewegung des Cursors unterstützt. Das Hauptprogramm dieses Editors sieht so aus:

```
program Editor;
uses Crt;
const
  X1  = 50;
  Y1  =  5;
  X2  = 75;
  Y2  = 22;
var
  Ch: Char;

< weitere Prozeduren >

begin
  CheckBreak  := False;           { kein Abbruch mit Ctrl-Break }
  CheckEOF    := False;           { ^Z nicht als Dateiende }
  DirectVideo := True;            { direkter Bildspeicher-Zugriff }
  Fensterrahmen(X1,Y1,X2,Y2);
  GotoXY(1,1);
  repeat                          { die Hauptschleife:       }
    Ch := ReadKey;                {         ein Zeichen lesen }
    if Ch <> #0                   {                           }
      then Tastendruck(Ch)        { normale Taste  }
      else Funktionstaste(ReadKey) { Funktionstaste }
  until Ch = ^Z;                  { Ende mit Ctrl-Z }
  Window(1,1,80,25);              { Fenster zurück }
  ClrScr                          { Bildschirm löschen }
end.                               { Aus. }
```

Nach dem Setzen der verschiedenen booleschen Variablen von *Crt* erzeugt das Programm über die Prozedur *Fensterrahmen* ein Textfenster auf dem Bildschirm; der nachfolgende Aufruf von *GotoXY* setzt den Cursor in die linke obere Ecke dieses Fensters.

Die Hauptschleife des Programms verwendet die (in *Crt* deklarierte) Funktion *ReadKey* zur direkten Abfrage der Tastatur. Wenn der Benutzer ein »normales« Zeichen eingegeben hat, liefert *ReadKey* dieses Zeichen direkt zurück, und es folgt ein Aufruf der Prozedur *Tastendruck*. Wurde dagegen eine Funktionstaste gedrückt, dann liefert *ReadKey* als erstes ein NUL-Zeichen (ASCII-Code 0), der zweite Aufruf liefert den »Scancode« der Funktionstaste, der über die Prozedur

Funktionstaste ausgewertet wird. Die Eingabe von Ctrl Z beendet das Programm (und setzt das Bildschirmfenster wieder auf die volle Größe).

Die Prozedur *Fensterrahmen* verwendet verschiedene Grafikzeichen des IBM PC, um einen Rahmen auf dem Bildschirm zu zeichnen, bevor das eigentliche Textfenster über die Routine *Window* aus der Unit *Crt* gesetzt wird. Sie kommt im Programm an die mit `< weitere Prozeduren >` gekennzeichnete Stelle:

```
procedure Fensterrahmen(X1,Y1,X2,Y2: Integer);
const
  EckobenLinks   = #201;
  LinieHor       = #205;
  EckobenRechts  = #187;
  LinieVert      = #186;
  EckuntenLinks  = #200;
  EckuntenRechts = #188;
var
  I: Integer;

begin
  Window(X1-1,Y1-1,X2+1,Y2+1);
  ClrScr;
  Window(1,1,80,25);
  GotoXY(X1-1,Y1-1);
  Write(EckobenLinks);
  for I := X1 to X2 do  Write(LinieHor);
  Write(EckobenRechts);
  for I := Y1 to Y2 do begin
    GoToXY(X1-1,I); Write(LinieVert);
    GotoXY(X2+1,I); Write(LinieVert)
  end;
  GotoXY(X1-1,Y2+1);
  Write(EckuntenLinks);
  for I := X1 to X2 do  Write(LinieHor);
  Write(EckuntenRechts);
  Window(X1,Y1,X2,Y2)
end;  { der Prozedur Fensterrahmen }
```

Die letzten beiden Prozeduren des Programms – *Tastendruck* und *Funktionstaste* – können entweder vor oder hinter *Fensterrahmen* stehen. *Funktionstaste* reagiert in unserem Beispiel nur auf die Pfeiltasten, Enter und Backspace – alle anderen Tasten haben einen Piepton zur Folge. Die Prozedur *Fensterrand* begrenzt die möglichen Bewegungen des Cursors auf das vorgegebene Textfenster.

```
procedure Tastendruck(Ch: Char);  { für "normale" Zeichen }
const
    BEL =  #7;                    { Piep! }
    BS  =  #8;                    { Rücktaste }
    CR  = #13;                    { RETURN }
begin
```

```
  if Ch = BS         then Write(BS,SP,BS)   { BS     }
  else if Ch = CR    then Writeln           { RETURN }
  else if Ch >= ' '  then Write(Ch)         { kein Steuerzeichen }
  else if Ch <> ^Z   then Write(BEL)        { sonst: falls <> ^Z
                                              -> Piep! }
end;  { der Prozedur Tastendruck }

procedure Fensterrand(Links: Integer; var X:Integer; Rechts:Integer);
begin
  if X < Links
    then X := Links
  else if X > Rechts
    then X := Rechts
end;  { der Prozedur Fensterrand }

procedure SetXY(XNeu,YNeu: Integer);
begin
  Fensterrand(1,XNeu,(1+X2-X1));
  Fensterrand(1,YNeu,(1+Y2-Y1));
  GotoXY(XNeu,YNeu)
end;  { der Prozedur SetXY }

procedure Funktionstaste(Ch: Char);
const
     PfeilAuf     = #72;
     PfeilLinks   = #75;
     PfeilRechts  = #77;
     PfeilAb      = #80;
begin
  case Ch of
    PfeilAuf   : SetXY(WhereX,WhereY-1);
    PfeilLinks : SetXY(WhereX-1,WhereY);
    PfeilRechts: SetXY(WhereX+1,WhereY);
    PfeilAb    : SetXY(WhereX,WhereY+1)
  end
end;  { der Prozedur Funktionstaste }
```

Prozedur-Variablen anstelle von case

Prozedur-Variablen – auf die in Kapitel 8 »Prozeduren und Funktionen« des *Programmierhandbuches* noch näher eingegangen wird – stellen (nicht nur) zu **case**-Anweisungen eine interessante Alternative dar. Das folgende Beispiel soll lediglich die praktische Anwendung verdeutlichen – für unser Programm *Editor* ist es sicher ein wenig überdimensioniert:

```
program ProcPtrDemo;
uses Crt;
```

```
type     { prozeduraler Typ mit zwei Parametern }
 ProcPtr = procedure(x, y: Integer);

var
 CursorKeys: array  of ProcPtr; {für alle 256 möglichen Tasten }
 Ch: Char;
```

ProcPtrDemo deklariert einen Variablentyp für Prozeduren mit zwei Integer-Parametern sowie ein Array mit 256 Elementen dieses Typs.

```
procedure Dummy(XNeu, YNeu: Integer); far;
begin
  Writeln('Unbekannt!'#7);
end;

procedure PfeilAuf(XNeu, YNeu: Integer); far;  { #72 }
begin Writeln('PfeilAuf'); end;

procedure PfeilAb(XNeu, YNeu: Integer); far;  { #80 }
begin Writeln('PfeilAb'); end;

procedure PfeilLinks(XNeu, YNeu: Integer); far; { #75 }
begin Writeln('PfeilLinks'); end;

procedure PfeilRechts(XNeu, YNeu: Integer); far; { #77 }
begin Writeln('PfeilRechts'); end;
```

Hier sind nur die vier Routinen skizziert, die auch von der Prozedur *Funktionstaste* des Programms *Editor* unterstützt werden. Zur Laufzeit des Programms werden die Adressen von *PfeilAuf..PfeilRechts* in den Elementen von *CursorKeys* gespeichert – der Aufruf der Routinen erfolgt mit dem von der Tastatur gelesenen Zeichen als *Index*.

```
procedure InitCursorKeys;
var InitCh: Char;

begin        { Alle Elemente von CursorKeys auf die Routine Dummy }
  for InitCh := #0 to #255 do
    CursorKeys[InitCh] := Dummy;

  CursorKeys[#72] := PfeilAuf;   { und dann die definierten Bear- }
  CursorKeys[#75] := PfeilLinks; { beitungsroutinen eintragen }
  CursorKeys[#77] := PfeilRechts;
  CursorKeys[#80] := PfeilAb;
end;
```

InitCursorKeys setzt zuerst sämtliche Elemente von *CursorKeys* auf die Routine *Dummy* (die ihrerseits nur mit der Ausgabe `Unbekannt!` reagiert). Danach werden die tatsächlich definierten Behandlungsroutinen für Tastendrücke eingetragen.

```
begin
  InitCursorKeys;  { Initialisierung von CursorKeys }
  repeat
    Ch:= ReadKey;
```

```
    if Ch <> #0 then Write(Ch)
     else CursorKeys[ReadKey](0,0);  { Bearbeitung von
                                       Funktionstasten }
  until Ch = #27;
end.
```

Nach diesen Vorbereitungen fällt die Bearbeitung der Funktionstasten wirklich verblüffend einfach aus: Wenn *ReadKey* beim ersten Aufruf das Ergebnis *#0* liefert, folgt ein *Aufruf* einer der Routinen, die in *CursorKeys* gespeichert sind, wobei das Programm den Scancode der Funktionstaste als Index benutzt. Da *CursorKeys* für jeden möglichen Scancode ein eigenes Element definiert (das in unserem Beispiel in den meisten Fällen auf die Prozedur *Dummy* zeigt), ist nicht einmal eine Bereichsprüfung des Scancodes notwendig.

Weitere Details zu prozeduralen Variablen finden Sie (wie bereits erwähnt) in Kapitel 8 »Prozeduren und Funktionen« des *Programmierhandbuches*.

5. Die Verwaltung von Projekten

Mittlerweile sind wir von »Einzeilern« über Standard-Units bis zu selbstgeschriebenen Units vorgestoßen und so in der Lage, große modulare Programme zu schreiben, die sich aus Dutzenden von Quelltexten zusammensetzen. Damit sind wir gleichzeitig beim zentralen Problem jeder modularen Programmiersprache angekommen: Je größer die Anzahl der Dateien wird, desto mehr steigt auch der dafür benötigte Verwaltungsaufwand.

Bei vielen Compilern muß der Programmierer die Frage »Wenn ich Modul A verändere, muß ich dann Modul B und D neu compilieren – oder vielleicht Modul F?« selbst beantworten. Turbo Pascal verfügt dagegen über eine *Make*-Funktion, die die Versionsalter aller Teile eines Programms automatisch berücksichtigt und derartige Überlegungen unnötig macht.

Zum Thema Programmorganisation

Wir haben schon öfters erwähnt, daß über das Unit-Konzept Programme mit mehr als 64 KByte Code möglich sind, weil jede Unit ein eigenes Segment zur Verfügung hat. Wenn Sie *Graph* und andere umfangreiche Units aufnehmen, geschieht die Unterteilung automatisch (und Ihr Programm hat nach wie vor *eigene* 64 KByte Platz zur Verfügung). Wie aber organisiert man ein großes Programm selbst in Units? Der erste Schritt besteht darin, alle globalen Datendefinitionen (Konstanten, Datentypen und Variablen) in einer eigenen Unit zusammenzufassen. Das nächste Beispiel zeigt, warum dies notwendig ist:

```
var  Puffer: string;    { eine globale Variable }

procedure Proc_1;
begin
  Readln(Puffer);       { ... weitere Operationen }
end;

procedure Proc_2;
begin
  Writeln(Puffer);      { ... weitere Operationen }
end;
```

In diesem Fragment greifen sowohl *Proc_1* als auch *Proc_2* auf die globale Variable *Puffer* zu. Damit diese Variable auch dann noch beiden Prozeduren zur Verfügung steht, wenn wir *Proc_1* und *Proc_2* in voneinander getrennten Units deklarieren, erzeugen wir eine Unit namens *Global*, deren Interface-Teil die Deklaration von *Puffer* enthält:

```
unit Global;
interface
var
  Puffer: string;
...     { weitere globale Deklarationen }
implementation
        { der Implementations-Teil ist leer! }
begin
        { Hier könnten Initialisierungen der globalen
          Variablen programmiert sein }
end.
```

Diese Unit läßt sich nun mit **uses** in alle anderen Units des Programms aufnehmen, also

```
unit Teil_1;
interface
  uses Global;
  procedure Proc_1;

implementation
  ...      { Definition von Proc_1 ...}
end.
```

Hinweis: Die im Interface-Teil einer Unit deklarierten Bezeichner stehen jedem Programm oder jeder Unit zur Verfügung, das diese Unit mit **uses** aufnimmt. Umgekehrt ist das dagegen nicht der Fall: Aus der Unit heraus kann nicht auf Bezeichner des Programms zugegriffen werden. (Wenn Sie an Include-Dateien gewöhnt sind, müssen Sie hier etwas umdenken).

Der nächste Schritt: Fassen Sie alle »Dienstroutinen« Ihres Programms in einer zweiten Unit zusammen. Unabhängig von der Art des Programms wird es immer eine Reihe von Routinen geben, die von fast allen Programmteilen aus benutzt werden und in keine eigene Kategorie fallen. Die entsprechende Unit hat in unserem Beispiel den Namen *Diverses* und verwendet die Unit *Global*:

```
unit Diverses;
interface
  uses Global;
  function HexToDec(x: string): Word;
  function DecToHex(x: Word): string;
  procedure Dir_Ausgabe(Name: string);
  function Datei_suchen (var Name: string): Boolean;
  ...
implementation
   ...
```

Die einzelnen Arbeitssschritte dieser Zusammenfassung können in etwa so aussehen:

- Laden Sie den Quelltext Ihres Programms (wobei Sie mit *File/Save as* eine Kopie für Not- und Unfälle erzeugen sollten).

- Verwenden Sie die Blockbearbeitungsfunktionen des Editors, um alle Routinen, die in der neuen Unit zusammengefaßt werden sollen, an das Ende dieses Quelltextes zu verschieben.

- Schreiben Sie die »zusammenkopierten« Routinen als Block unter dem Namen der zukünftigen Unit (für unser Beispiel: DIVERSES.PAS) auf die Diskette, und löschen Sie den Block im Editor.

- Fügen Sie hinter der **program**-Deklaration die Zeile

```
uses Diverses;
```

ein und speichern Sie den Quelltext.

- Laden Sie die als Block geschriebene Datei (DIVERSES.PAS) und stellen Sie diesem Text die folgenden Zeilen voran:

```
unit Diverses;    <- derselbe Name, der im Hauptprogramm
                     mit uses angegeben wurde
interface
uses Global;
                  <- Platz für Funktions- und Prozedur-Deklarationen
implementation
                  <- hier sollten die Prozeduren und Funktionen
                     stehen
```

- Kopieren Sie die Deklarationen aller Funktionen und Prozeduren zeilenweise in den Raum zwischen **interface** und **implementation**.

- Fügen Sie ein abschließendes **end.** in die letzte Zeile des Quelltextes ein.

- Prüfen Sie mit *Compile/Compile*, ob der Unit-Quelltext frei von syntaktischen Fehlern ist. Nach eventuellen Korrekturen speichern Sie den Quelltext auf der Diskette.

- Laden Sie den (gekürzten) Quelltext des Hauptprogramms und versuchen Sie eine Compilierung via *Compile/Make*.

In einem weiteren Schritt können Sie sich daran machen, die restlichen Routinen Ihres Programms in einzelne Funktionsgruppen zu gliedern und dafür entsprechende Units zu erzeugen, die ihrerseits *Global* und *Diverses* mit **uses** aufnehmen. Für einen »typischen« Editor bieten sich beispielsweise Funktionsgruppen wie »Dateilesen und -schreiben«, »Bildschirmsteuerung«, »Formatierung«, »Eingabe« und »Ausdrucken« an.

Initialisierung

Wie in Kapitel 3 »Units, Overlays und andere Geheimnisse« besprochen (und zuvor bei der Unit *Global* noch einmal angedeutet), kann jede Unit einen eigenen Initialisierungs-Teil definieren, dessen Anweisungen vor dem Start des eigentlichen Programms ausgeführt werden. Die Reihenfolge des Aufrufs dieser Initialisierungs-Teile ist durch die **uses**-Anweisung im Hauptprogramm festgelegt. Bei einer Angabe wie

```
program Editor;
uses
  Global, Diverses, Eingabe, Format, Drucker;
```

wird also vor dem Start von *Editor* zuerst der Initialisierungs-Teil der Unit *Global* aufgerufen, danach der von *Diverses*, gefolgt von dem von *Eingabe* usw. (diese Aufrufe finden natürlich nur statt, wenn entsprechende Initialisierungs-Teile vorhanden sind).

Hinweis: Wie ebenfalls in Kapitel 3 »Units, Overlays und andere Geheimnisse« erläutert, verändert der Compiler die Reihenfolge der Aufrufe notfalls automatisch – ein eventueller Initialisierungs-Teil von *Global* wird deshalb als erstes aufgerufen, weil wir hier davon ausgehen, daß diese Unit von allen anderen Units des Programms vorausgesetzt wird und selber keine weiteren Units mit **uses** aufführt.

Die Menüpunkte Build und Make

Anhand des Beispiels *Editor* können wir nun konkret verdeutlichen, was es mit der »Verwaltung von Projekten« eigentlich auf sich hat. Der Übersichtlichkeit halber beschränken wir uns auf das Hauptprogramm *Editor*, die Deklarationen in *Global*, die Routinen in *Diverses* und das Modul *Eingabe*. Die **uses**-Anweisungen dieser vier Module lassen sich folgendermaßen darstellen:

```
unit Global;
 { keine uses-Anweisung }

unit Diverses;
  uses Global;
  unit Global;

unit Eingabe;
  uses Global,   Diverses;
  unit Global;   unit Diverses;
                   uses Global;
                   unit Global;

program Editor;
  uses Global,   Diverses,        Eingabe;
```

```
unit Global;   unit Diverses;   uses Global,   Diverses;
                 uses Global;   unit Global;   unit Diverses;
                 unit Global;                    uses Global;
                                                 unit Global;
```

Woraus folgt: Nach Änderungen innerhalb des Hauptprogramms muß nur *Editor* neu compiliert werden. Änderungen innerhalb von *Eingabe* erfordern eine erneute Compilierung von *Eingabe* und *Editor*, Änderungen in *Diverses* machen die Neucompilierung von *Diverses*, *Eingabe* und *Editor* notwendig; falls Sie etwas in *Global* verändern, müssen Sie alle vier Module des Programms neu übersetzen lassen.

Natürlich läßt sich diese Logik auch über Bord werfen, indem wir einfach jedesmal *alle* Module neu compilieren. Damit wären wir allerdings wieder beim Prinzip der Include-Dateien angelangt – es läßt sich leicht ausrechnen, ab welcher Projektgröße auch mit der schnellsten Festplatte nach jeder kleinen Änderung eine längere Kaffepause notwendig ist...

Die Funktion Make

ist (wie Sie wohl schon erraten haben) die Lösung dieser Probleme. Unabhängig davon, welche Teile eines Programms Sie verändern – der Start des Compilers über *C/Make* oder F9 bzw. zusammen mit dem Kommandozeilen-Parameter /M erzeugt ein Programm, dessen einzelne Teile immer auf dem neuesten Stand sind. *Make* führt drei verschiedene Prüfungen aus:

- Zuerst werden Uhrzeit und Datum aller beteiligten .TPU-Dateien mit den entsprechenden Quelltexten verglichen. Wenn ein Unit-Quelltext (d.h. die .PAS-Datei) seit der letzten Compilierung verändert wurde, dann ist er neueren Datums als die .TPU-Datei und wird von *Make* automatisch recompiliert. Dasselbe ist der Fall, wenn die .TPU-Datei überhaupt noch nicht existiert bzw. von *Make* nicht gefunden werden konnte.

- Die zweite Prüfung betrifft den Interface-Teil beteiligter Units. Turbo Pascal bildet für jedes Interface eine Prüfsumme, die in allen Units gespeichert wird, die diese Unit benutzen. In unserem Beispiel ist also die Prüfsumme von *Diverses* in *Editor* und *Eingabe* gespeichert – wenn dieser gespeicherte Wert nicht mehr mit dem von *Diverses* übereinstimmt, dann werden *Editor* und *Eingabe* neu compiliert.

- Die dritte Prüfung betrifft Include- und .OBJ-Dateien: Wenn eine via {$I} oder {$L} angegebene Datei neueren Datums als der Pascal-Text ist, in dem sich dieser Compiler-Befehl befindet, dann nimmt *Make* eine erneute Compilierung der Pascal-Datei vor. Auf diese Weise wird sichergestellt, daß ein Programm oder eine Unit auch dann auf dem neuesten Stand ist, wenn eine dazugehörige Include-Datei verändert oder eine .OBJ-Datei neu assembliert wurde.

Hinweise: Der zweite Vergleich ist unabhängig vom ersten und dritten. Wenn die Veränderung nur innerhalb des Implementations- oder Initialisierungs-Teils

einer Unit stattfindet, ist eine Recompilierung weiterer Programmteile nicht notwendig und wird deshalb auch nicht ausgeführt. Wenn *Make* den zu einer Unit gehörigen Quelltext nicht findet, dann wird diese Unit automatisch als »auf dem neuesten Stand« angenommen und nicht weiter geprüft. (*Make* versucht also in keinem Fall, die Standard-Units aus TURBO.TPL neu zu compilieren).

Die Funktion Build

Ein Aufruf des Compilers über *C/Build* bzw. den Kommandozeilen-Parameter /B arbeitet nach derselben Baumstruktur wie *Make* und erfaßt also auch alle an einem Programm beteiligten Module (außer den Units in TURBO.TPL). Ein Vergleich von Datum und Uhrzeit oder Prüfsummen findet im Gegensatz zu *Make* nicht statt – *Build* recompiliert einfach *sämtliche* Dateien des Programms, deren Quelltexte verfügbar sind.

Da *Build* eigentlich nur im Zusammenhang mit bedingter Compilierung verwendet wird, leisten wir uns einen Vorgriff auf die ausführliche Besprechung in Kapitel 2 »Compiler-Befehle« des *Referenzhandbuches* und erläutern kurz die zugrundeliegenden Prinzipien.

Bedingte Compilierung

Turbo Pascal stellt zwei Konstrukte zur Verfügung, mit denen Teile eines Quelltextes von der Compilierung ausgeschlossen bzw. in eine Compilierung einbezogen werden können:

```
{$IFDEF Symbolname} { *** 1. Konstrukt *** }
   ...
  < Pascal-Anweisungen und Deklarationen ä werden
    compiliert, wenn Symbolname definiert ist >
   ....
{$ENDIF}
{$IFDEF Symbolname} { *** 2. Konstrukt - Erweiterung des ersten *** }
   ...
  < Pascal-Anweisungen und Deklarationen - werden
    compiliert, wenn Symbolname definiert ist >
   ....
{$ELSE}
  ...
< Dieser Teil wird compiliert, wenn Symbolname
  nicht definiert ist >
 ...
{$ENDIF}
```

Symbole zur bedingten Compilierung haben keinen Wert im eigentlichen Sinne – sie sind entweder definiert oder existieren nicht. Eine Prüfung mit {$IFDEF} (= »if defined«) ergibt ein Äquivalent zu *True*, wenn der angegebene

Symbolname definiert ist; ansonsten ist das Ergebnis *False*. Programmteile, die mit Befehlen zur bedingten Compilierung eingeschlossen sind, können beliebige Anweisungen und Compiler-Befehle enthalten.

Die Definition bedingter Symbole kann mit dem Compiler-Befehl {$DEFINE} innerhalb des Quelltextes oder über *O/C/Conditional defines* bzw. den Kommandozeilen-Parameter /D geschehen. Ein Beispiel dazu:

```
program DiskUtil;   { Datei: DISKUTIL.PAS }
{$IFDEF LAPTOP}
  {$I Disk3-2.PAS}  { Include-Datei mit Definitionen für
                      3 1/2"-Laufwerke }
{$ELSE}
  {$I Disk5-4.PAS}  { Include-Datei mit Definitionen für
                      5 1/4"-Laufwerke }
{$ENDIF}
  ...
```

Wenn Sie das Symbol LAPTOP bei der Compilierung definieren, also z.B. die Kommandozeilen-Version des Compilers mit

```
tpc /dLAPTOP diskutil
```

aufrufen, dann sucht der Compiler nach der Include-Datei DISK3-2.PAS und nimmt sie in den Quelltext auf, ansonsten wird mit DISK4-5.PAS gearbeitet.

Die bedingte Compilierung macht es (unter anderem) möglich, für verschiedene Computermodelle (Laufwerke, Festplatte, Video-Adapter, Coprozessor...) mit einem einzigen Quelltext auszukommen – anstelle entsprechender Veränderungen des Quelltextes werden vor der Compilierung lediglich jeweils die entsprechenden Symbole definiert.

Nun sollte auch klar sein, wann *Build* eingesetzt werden muß: Bei einer reinen Redefinition von Symbolen über *O/C/Conditional defines* bzw. den Parameter /D der Kommandozeilen-Version bleiben die Quelltext-Dateien unverändert – der Uhrzeit- und Datumsvergleich von *Make* greift in diesem Fall nicht mehr.

Hinweis: Natürlich bietet Turbo Pascal noch eine Reihe weiterer Möglichkeiten zusammen mit bedingten Symbolen, {$IFDEF} und {$ENDIF} sind auch bei weitem nicht die einzigen Compiler-Befehle. Wir verweisen auf Kapitel 2 »Compiler-Befehle« des *Referenzhandbuches*.

Das Programm MAKE.EXE

Solange Sie ausschließlich mit Pascal-Quelltexten arbeiten, werden Sie auf dieses Programm wahrscheinlich nie zurückgreifen müssen – wie aber sieht es bei der Verwendung von Assembler-Modulen, C-Modulen oder anderen Daten aus, die Turbo Pascal (aus verständlichen Gründen) nicht selbst auf den neuesten Stand bringen kann? Anders gesagt: Wenn das Pascal-Unit *Eingabe* aus unserem Beispiel *Editor* die Datei INPUT1.OBJ mit aufnimmt, kann die eingebaute *Make*-Funktion von Turbo Pascal zwar prüfen, ob INPUT1.OBJ neueren

Datums als EINGABE.TPU ist – wenn Sie aber vergessen haben, INPUT1.OBJ nach einer Veränderung neu zu assemblieren, dann ist *Make* machtlos.

Das liegt daran, daß *Make* nur zwei *implizite Abhängigkeiten* und eine einzige Reaktionsmöglichkeit kennt:

```
XXXX.TPU    <--- wird gegen XXXX.PAS geprüft.
                 Wenn XXXX.PAS neueren Datums ist, dann
                 --> Compilierung von XXXX.PAS

XXXX.EXE    <--- wird gegen XXXX.PAS geprüft.
                 Wenn XXXX.PAS neueren Datums ist, dann
                 --> Compilierung von XXXX.PAS
```

MAKE.EXE verwendet dieselbe Logik wie *Make*, was die Vergleiche von Uhrzeit und Datum betrifft, hebt dabei aber alle Grenzen radikal auf. Eine Definition für MAKE.EXE hat in etwa die folgende Form:

```
XXXX.XYZ   <---- wird gegen eine beliebige andere Datei geprüft.
                 Wenn diese andere Datei neueren Datums ist, dann
                 --> DOS-Befehl
                     DOS-Befehl
                          ....
```

Welche Dateien Sie nun mit MAKE.EXE gegeneinander prüfen und welche DOS-Befehle Sie als Folge dieser Prüfung ausführen lassen, liegt hundertprozentig bei Ihnen. Normalerweise werden diese DOS-Befehle in Aufrufen des Compilers, eines Assemblers usw. bestehen – niemand hält Sie aber beispielsweise davon ab, auf das Fehlen einer Datei mit einer Ausgabe auf den Drucker und dem anschließenden Start eines Spielprogramms zu reagieren.

Eine vollständige Dokumentation von MAKE.EXE finden Sie in Kapitel 10 »Die Zusatzprogramme«. Wir wollen hier nur eine der einfacheren Verwendungsmöglichkeiten zeigen.

Ein Beispiel

Nehmen wir noch einmal unser Programm *Editor* und setzen wir voraus, daß die Unit *Eingabe* zwei Unterprogramme in Maschinensprache enthält, die mit einem Assembler erstellt wurden. Die entsprechenden Dateien sollen die Namen INPUT1.ASM, INPUT1.OBJ, INPUT2.ASM und INPUT2.OBJ tragen und innerhalb des Interface-Teils von *Eingabe* deklariert sein. Damit es nicht zu einfach wird, verwenden beide die Variable *Puffer* der Unit *Global*.

Zwischen den Pascal-Quelltexten und .TPU-Dateien bzw. Assembler-Quelltexten und .OBJ-Dateien bestehen also die folgenden Zusammenhänge:

```
GLOBAL.TPU      <- Neucompilierung bei Änderung von GLOBAL.PAS
DIVERSES.TPU    <- dito bei Änderung von DIVERSES.PAS
EINGABE.TPU     <- dito bei Änderung von EINGABE.PAS
EDITOR.EXE      <- dito bei Änderung von EDITOR.PAS
```

```
INPUT1.OBJ     <- Neu-Assemblierung bei Änderung von INPUT1.ASM
INPUT2.OBJ     <- dito bei Änderung von INPUT2.ASM
```

Hier haben wir dieselben Regeln für die Assembler-Module wie für Pascal-Dateien: Wenn der Quelltext neueren Datums als das Ergebnis der Übersetzung ist, muß erneut übersetzt werden – nur daß es sich hier nicht um einen Aufruf des Compilers, sondern um einen Aufruf eines Assemblers handelt.

Außer den Abhängigkeiten der Pascal-Dateien untereinander, die wir bereits besprochen haben (*Diverses* hängt vom Interface-Teil von *Global* ab usw), kommen noch zwei *explizite* Regeln hinzu:

```
INPUT1.OBJ     <- Neu-Assemblierung bei Änderung von GLOBAL.PAS/.TPU
INPUT2.OBJ     <- dito
EINGABE.TPU    <- Neucompilierung bei Änderung von INPUT1.OBJ
                  und / oder INPUT2.OBJ
```

Eine Datei für MAKE

MAKE.EXE erwartet ein Regelwerk wie das zuvor beschriebene in einer ASCII-Datei, die in unserem Beispiel so aussieht:

```
editor.exe: global.pas diverses.pas eingabe.tpu
  tpc editor /m

eingabe.tpu: eingabe.pas input1.obj input2.obj
  tpc eingabe /m

input1.obj: input1.asm global.pas
  tasm input1;

input2.obj: input2.am global.pas
  tasm input2;
```

Diese Regeln stehen für die folgenden Zusammenhänge:

- EDITOR.EXE ist von den Dateien GLOBAL.PAS, DIVERSES.PAS und EINGABE.TPU abhängig. Wenn auch nur eine dieser Dateien neueren Datums als EDITOR.EXE ist, dann wird die Kommandozeilen-Version des Compilers (mit tpc editor /M) aufgerufen. Der Parameter /M aktiviert die eingebaute *Make*-Funktion des Compilers, die sich um alle impliziten Abhängigkeiten und die Zusammenhänge der Units untereinander kümmert.

  Anstelle von TPC läßt sich an jeder Stelle einer MAKE-Datei auch die integrierte Entwicklungsumgebung aufrufen (vgl. »Kommandozeilen-Parameter« in Kapitel 7 »Die IDE-Referenz«). Parameter und Dateinamen sind dieselben; der Befehl für eine Compilierung von *Editor* via TURBO.EXE wäre also:

```
turbo editor /M
```

- EINGABE.TPU ist von den Dateien EINGABE.PAS, INPUT1.OBJ und INPUT2.OBJ abhängig. Hier gilt dasselbe Prinzip wie bei EDITOR.EXE: Wenn auch nur eine der angegebenen Dateien neueren Datums als EINGABE.TPU selbst ist, wird TPC aufgerufen und nimmt eine neue Compilierung vor.

- Analoges gilt für die Dateien INPUT1.OBJ und INPUT2.OBJ, die von ihren

Quelltexten und von GLOBAL.PAS abhängig sind. Hier wird für eine erneute Übersetzung allerdings nicht der Compiler, sondern der Assembler aufgerufen.

MAKE.EXE ist in Kapitel 10 »Die Zusatzprogramme« beschrieben. Wir lassen deshalb an dieser Stelle Formales außer acht und konzentrieren uns auf das logische Prinzip der verwendeten Regeln:

```
Ziel-Datei: Quell-Datei Quell-Datei Quell-Datei ...
  DOS-Befehl
  DOS-Befehl
  ...
```

MAKE.EXE vergleicht Datum und Uhrzeit der *Ziel-Datei* mit allen angegebene *Quell-Dateien*. Wenn auch nur eine der *Quell-Dateien* neueren Datums als die *Ziel-Datei* ist, werden die angegebenen *DOS-Befehle* der Reihe nach ausgeführt.

Alle Vergleichen mit *Ziel-Dateien* liegt eine Baumstruktur zugrunde: Bevor der Vergleich mit einer *Quell-Datei* tatsächlich stattfindet, wird geprüft, ob für diese *Quell-Datei* eine eigene Regel vorhanden ist – wenn ja, wird diese Regel zuerst bearbeitet. In unserem Beispiel ergeben sich deshalb die folgenden Vergleiche:

```
1. INPUT1.OBJ   mit GLOBAL.PAS
2. INPUT2.OBJ   mit GLOBAL.PAS
3. EINGABE.TPU  mit EINGABE.PAS, INPUT1.OBJ, INPUT2.OBJ
4. EDITOR.EXE   mit allen angegebenen Dateien
```

Ein *Hinweis* für Spezialisten: Unser Beispiel hat einen kleinen Schönheitsfehler – INPUT1 und INPUT2 werden bei *jeder* Änderung von GLOBAL.PAS neu assembliert (und nicht nur bei Veränderungen des Interface-Teils).

Der Aufruf von MAKE

Nehmen wir der Vollständigkeit halber einmal an, Sie hätten die zuvor gezeigten Regeln mit dem Editor der integrierten Entwicklungsumgebung in eine Datei namens EDITOR.MAK eingegeben. Der Aufruf von MAKE erfolgt danach von der DOS-Kommandoebene aus mit dem Befehl

```
make -fEDITOR.MAK
```

Nicht nur, daß MAKE.EXE noch eine Reihe weiterer Kommandozeilen-Parameter kennt – wir haben hier auch nur einen Bruchteil der Möglichkeiten dieses Programms gezeigt. Details zu Dingen wie Symbolen, bedingter Ausführung innerhalb der MAKE-Datei (nach demselben Prinzip wie **if.. then.. else**), arithmetischen Operatoren, der Berechnung von Ausdrücken usw. usf. finden Sie in Kapitel 10 »Die Zusatzprogramme«.

6. Fehlersuche und der integrierte Debugger

Der Originalterminus *debugging* (der im Deutschen mit »Fehlersuche und -bereinigung« nur ein annäherndes Äquivalent besitzt) entstand vor rund 35 Jahren. Damals stellte sich eine tote Motte (»Bug«) nach tagelanger Suche als Fehlerursache heraus – sie war zwischen den Kontakten eines Relais eingeklemmt. Relais und Röhren sind inzwischen recht selten geworden, was man von Motten und ihren Äquivalenten in der Computertechnik leider nicht sagen kann.

Wenn Sie noch nie einen Fehler erzeugt haben, dann haben Sie auch noch nie programmiert. Fehler sind einfach eine unvermeidbare Begleiterscheinung jeder Programmierarbeit – noch dazu müssen sie in den allermeisten Fällen durch schlichtes Ausprobieren und Neuversuchen (Trial and Error) beseitigt werden.

Allerdings: mit Turbo Pascal stehen Ihnen nicht nur die »klassischen« Methoden der Fehlersuche zur Verfügung (auf die wir in den nächsten Abschnitten kurz eingehen) – sondern auch ein wirklich schweres Geschütz: der *integrierte Debugger*, mit dem sich der weitaus größte Teil dieses Kapitels beschäftigt.

Fehlerklassen

Je nachdem, wie, wann und wo sich ein Fehler bei der Programmentwicklung bemerkbar macht, unterscheidet man drei Arten, auf die die nächsten Abschnitte kurz eingehen:

Compilierungsfehler

Der erste (und häufigste) Fehlertyp ist syntaktischer oder semantischer Natur und ruft bei der Compilierung eine entsprechende Meldung hervor. Sie haben vergessen, eine Variable zu deklarieren, rufen eine Prozedur oder Funktion mit falscher Parameterzahl auf oder haben sich vertippt... Anders gesagt: Der Compiler erhält eine Befehlsfolge, die nicht mit den Regeln von Pascal vereinbar ist. Pascal folgt außerordentlich strengen Regeln – selbst im Vergleich zu den meisten anderen Programmiersprachen. Erfreulich daran: In Pascal besteht der

größte Teil des Arbeitsaufwandes aus syntaktischen Korrekturen (die aufgrund der Meldungen des Compilers recht einfach auszuführen sind).

Turbo Pascal bricht die Compilierung bei der Erkennung des ersten Fehlers ab; Maschinencode wird erst dann erzeugt, wenn der Quelltext völlig fehlerfrei ist. Nach Abbruch der Compilierung wird der Editor aktiviert und der Cursor auf die entsprechende Stelle im Quelltext gesetzt, in der Statuszeile des Editors erscheint die Fehlermeldung, die beim ersten Tastendruck verschwindet (und sich jederzeit über Ctrl Q W zurückholen läßt). Nach einer entsprechenden Korrektur können Sie die Compilierung erneut starten.

Hinweis: Speziell bei der Arbeit mit Diskettenlaufwerken sollte der Schalter *Compile/Destination* zumindest solange auf *Memory* gesetzt sein, bis Ihr Programm frei von Syntaxfehlern ist.

Laufzeitfehler

Wie ihr Name bereits sagt, treten Fehler dieser Art während der Laufzeit des Programms auf – Sie haben also ein syntaktisch korrektes Programm compiliert, das bei seiner Ausführung etwas Unzulässiges unternimmt, wie beispielsweise eine Division durch Null oder den Versuch, eine nicht existierende Datei zu öffnen. In solchen Fällen erhalten Sie eine Meldung des Formats

```
Runtime error xxx at yyyy:zzzz
```

xxx steht hier für die Nummer des Fehlers, *yyyy:zzzz* für die (hexadezimale) Adresse des Befehls, der den Fehler ausgelöst hat.

Wenn Sie das Programm von der integrierten Entwicklungsumgebung aus gestartet haben, lokalisiert Turbo Pascal die entsprechende Stelle im Quelltext automatisch.

Haben Sie das Programm dagegen von der DOS-Kommandoebene aus gestartet, dann müssen Sie die als *yyyy:zzzz* angegebene Adresse notieren und die Funktion *Search/Find error* der integrierten Entwicklungsumgebung bzw. den Parameter `/F` der Kommandozeilen-Version des Compilers benutzen.

Hinweis: Diese beiden Möglichkeiten sind in Kapitel 7 »Die IDE-Referenz« bzw. in Kapitel 9 »Kommandozeilen-Version« eingehend erläutert.

Laufzeitfehler werden im Prinzip wie Syntaxfehler korrigiert: Ändern Sie Ihren Quelltext entsprechend, und fügen Sie eventuell notwendige Prüfungen hinzu. Danach können Sie den Compiler erneut aufrufen und das veränderte Programm starten.

Logische Fehler

sind die dritte, letzte und am schwierigsten zu beseitigende Fehlerart: sie machen sich normalerweise dadurch bemerkbar, daß ein (syntaktisch korrektes) Programm ordentlich bis zu seinem regulären Ende durchgeführt wird – nur macht es leider dabei etwas anderes, als der Programmierer im Sinn hatte.

So vielfältig wie die möglichen Gründe (nicht initialisierte Variablen, auf-

summierte Rundungs- und/oder Denkfehler) sind auch die Ergebnisse: Grafikbilder, bei denen die unterste Zeile verschwindet; Dateien, die vom Programm als »fehlerfrei gespeichert« gemeldet werden, aber rätselhafterweise drei Bytes zuviel oder einen Datensatz zuwenig haben; Tastenbefehle, die eine augenscheinlich korrekt programmierte Eingaberoutine einfach völlig ignoriert...

Einen Trost haben wir für Sie: bei dieser Fehlerklasse hilft Ihnen der integrierte Debugger am meisten.

Klassische Methoden der Fehlersuche

Im folgenden finden Sie einen kurzen Überblick über die klassischen Methoden der Fehlersuche.

Fehlerprüfungen der Ein-/Ausgabe

Gehen wir einmal davon aus, daß Ihr Programm die Eingabe zweier Integerzahlen erwartet, Sie aber nicht-numerische Zeichen (Buchstaben, Kommas etc.) eingeben. Das Resultat ist bekannt – das Programm wird mit einem Laufzeitfehler abgebrochen.

Turbo Pascal erlaubt das zeitweilige Abschalten der eingebauten Prüfung für Ein- und Ausgabefehler über die Compiler-Befehle {$I+} und {$I-}. Die Standardvorgabe des Compilers ist {$I+}, d.h. Ein- und Ausgaben werden automatisch überprüft. Innerhalb eines Programmteils, dem der Compiler-Befehl {$I-} vorangestellt ist, lassen sich Fehlerbedingungen durch Pascal-Befehle prüfen und abfangen. Ein Beispielprogramm:

```
programNummer3;    {Version 4, mit Hilfsvariable}

var
  A, B, C: Integer;
  Fehlercode: Integer;

begin
  {$I-}            {schaltet die Überprüfung ab}
  repeat
    Write ('Geben Sie zwei Werte ein: ');
    Readln (A, B);
    Fehlercode:= IOResult
    if Fehlercode <> 0 then
       Writeln ('Fehlerhafte Eingabe!');
  Until Fehlercode = 0;
  {$I+}            {Überprüfung wieder an}

  if B = 0
    then Writeln('Division durch 0 nicht zulässig!')
```

```
    else begin
     C := A div B;
     Writeln ('Das Ergebnis ist: ',C);
    end;
end.
```

Das Ergebnis von *IOResult* muß in einer Variablen gespeichert werden, weil wir diesen Wert sowohl für die **if**-Bedingung als auch für die Prüfung durch **until** verwenden und *IOResult* den Fehlerstatus nach jedem Aufruf automatisch zurücksetzt.

Prüfungen bei der Eröffnung einer Datei lassen sich auf ähnliche Weise über die Compiler-Befehle {$I-} und {$I+} zusammen mit *IOResult* programmgesteuert ausführen:

```
program DateiTest;
var
  Name : string;
  f : Text;
  Fehlercode: Integer;
begin
  repeat
    Write('Geben Sie den Dateinamen ein: ');
    Readln(Name);
    Assign(f, Name);
    {$I-} Reset(f); {$I+}
    Fehlercode := IOResult;
    if Fehlercode <> 0
     then Writeln('Datei ', Name, ' nicht gefunden!');
  until Fehlercode = 0;
  ....
```

Bereichs-überschreitungen

In diese Fehlerkategorie fallen Zuweisungen an Variablen, die außerhalb des gültigen Wertebereiches liegen, unzulässige Indizierungen von Arrays und die Übergabe ungültiger Werte als Parameter. Das folgende Programm ist ein typisches Beispiel für einen kleinen Denkfehler, der normalerweise einen Neustart des Systems erzwingt:

```
program FalscherBereich;
{ ACHTUNG: Dieses Programm darf nur von der integrierten
 Entwicklungsumgebung aus gestartet werden! }

var
  i : Byte;
  Summe: LongInt;

begin
```

```
  { sollte die Gesamtsumme von 99 + 97 + 95... + 3 + 1 berechnen }
  Summe := 0;
  i := 99;
  repeat
    Summe := Summe + i;
    i := i - 2;
  until i = 0;
  Writeln('Die Summe ist: ', Summe);
end.
```

Hier ist die Abbruchbedingung der Schleife nie erfüllt: *i* bekommt nacheinander die Werte 99, 97, 95... 5, 3, 1 und -1 zugewiesen. Da der Wert -1 für Variablen des Typs *Byte* nicht definiert ist und intern genauso wie der Wert 255 gespeichert wird, fängt die Zählung immer wieder »von vorne« an.

Vorsicht: Sie *können* dieses Programm ruhig ausprobieren – aber nur innerhalb der integrierten Entwicklungsumgebung (die auch bei Endlosschleifen noch einen Abbruch über Ctrl Break ermöglicht). Wenn Sie *FalscherBereich* dagegen als .EXE-Datei compilieren und von der DOS-Kommandoebene aus starten, hilft nur noch der Griff zum RESET-Knopf (ersatzweise dem Netzschalter).

Über einen Compiler-Befehl lassen sich derartige Fehler entdecken (die in realen Anwendungen meist wesentlich besser versteckt sind): In Programmteile, die mit {$R+} und {$R-} eingeklammert sind, fügt der Compiler für jede Veränderung von Integer-Variablen sowie für Indizierungen von Arrays zusätzlichen Prüfcode ein, der das Programm bei Überschreitungen mit einer entsprechenden Meldung abbricht.

Sie können die Bereichsüberprüfung für ein gesamtes Programm aktivieren, indem Sie einfach den Compiler-Befehl {$R+} an den Anfang des Quelltextes setzen. Allerdings wird dadurch für alle Operationen zusätzlicher Code erzeugt, der das Programm langsamer und umfangreicher macht. Es kann sogar Situationen geben, in denen Sie Bereichsgrenzen absichtlich überschreiten wollen oder müssen, z.B. bei der Arbeit mit dynamisch erzeugten Arrays oder bei der Verwendung von *Succ* und *Pred* zusammen mit Aufzählungstypen.

Fehlerstellen einkreisen

Zur Suche nach Fehlern logischer Natur, die nicht unbedingt zu einem Laufzeitfehler führen müssen, verwendet man in Pascal üblicherweise zusätzliche *Write*-Anweisungen, deren einziger Zweck die Verfolgung des Programmablaufs ist. (Dieses Verfahren ist nicht gerade elegant, aber wirksam). Ausgaben zur Fehlersuche lassen sich von einer globalen Variablen abhängig machen, die zur Verfolgung des Programmablaufs auf den Wert *True*, ansonsten auf den Wert *False* gesetzt wird.

Nehmen wir einmal an, Ihr Programm verändert zwei globale Variablen *A* und *B* über drei oder vier verschiedene Prozeduren. Am Programmende haben beide nicht die erwarteten Werte – wo sich der Fehler eingeschlichen hat, ist

nicht so einfach zu sagen. Zur Lokalisierung der Fehlerstelle verändern Sie alle dazugehörigen Prozeduren nach dem folgenden Schema:

```
procedure xxxx {(dazugehörige Parameter)}
{lokale Variablen und Deklarationen}
begin
  if Trace then Writeln ('Start von xxx: A = ', A, ', B = ',B);

  {Befehle und Anweisungen innerhalb der Prozedur}

  if Trace then Writeln ('Ende von xxx: A = ', A, ', B = ',B);
end; {Ende der Prozedur}
```

(Wir gehen hierbei davon aus, daß *Trace* eine globale boolsche Variable ist, die zu Beginn des Hauptprogramms auf *True* oder *False* gesetzt wurde).

Natürlich muß es sich bei *A* und *B* nicht unbedingt um globale Variablen handeln – denkbar wäre genauso eine Ausgabe übergebener Parameter. Nachdem zu irgendeinem Zeitpunkt nun falsche Werte für *A* und *B* ausgegeben werden, wissen Sie zumindest, welche Prozedur dafür verantwortlich ist. Wenn Sie nun die beiden »Trace«-Befehle dichter zusammenlegen, können Sie die Fehlerstelle einkreisen und gegebenenfalls korrigieren.

Mit einer zusätzlichen *Write*-Anweisung ließe sich auch der im Abschnitt »Bereichsüberschreitungen« gezeigte Fehler recht schnell lokalisieren:

```
program FalscherBereich;  { 2. Version mit Write-Befehl }
var
  i : Byte;
  Summe: LongInt;
begin {sollte die Gesamtsumme von 99 + 97 + 95... + 3 + 1 berechnen }
  Summe := 0;
  i := 99;
  repeat
    Summe := Summe + i;
    i := i - 2;          Write(i, '  ');  { neu eingefügt }
  until i = 0;
  Writeln('Die Summe ist: ', Summe);
end.
```

Diese Version von *FalscherBereich* gibt nicht nur nacheinander die Werte der Variablen *i* aus (wobei schnell zu erkennen ist, daß die Schleife immer wieder von vorne anfängt), sondern hat auch noch einen zweiten Vorteil: Das Programm läßt sich bei Ausgaben mit *Write* auch von der DOS-Kommandoebene aus über Ctrl Break abbrechen.

Der integrierte Debugger

Es gibt Fälle, in denen keine der zuvor gezeigten Methoden weiterhilft. Entweder läßt sich die Fehlerstelle einfach nicht lokalisieren – oder, genauso

unerfreulich: Sie haben einen Punkt bestimmt, können sich aber nicht erklären, was daran falsch sein sollte.

Wie Überschrift und Einleitung wohl bereits vermuten lassen, sind wir nun beim eigentlichen Thema dieses Kapitels angelangt: auf den nächsten Seiten geht es um die Menüs, Wahlpunkte und Möglichkeiten des Fehlerlokalisierungs- und Programmprüfprogramms der integrierten Entwicklungsumgebung – kurz: um den integrierten Debugger von Turbo Pascal.

Im ersten Teil finden Sie ein Tutorial zur schrittweisen Einführung in die Funktionsweise des Debuggers. Der zweite Teil besteht aus einer vollständigen Referenz, d.h. einer detaillierten Erläuterung sämtlicher Menüpunkte des Debuggers. Im dritten Teil schließlich geht es neben Grenzbedingungen (wie Platzproblemen im Hauptspeicher) um Richtlinien zur »defensiven Programmierung« – wir versuchen, einige allgemeine Strategien zu vermitteln, die die Notwendigkeit der späteren Fehlersuche zumindest reduzieren.

Teil I: Eine schrittweise Einführung

Da die Fülle der Suchmöglichkeiten und die Fälle möglicherweise gesuchter Fehler womöglich den Fehlersuchenden überfüllen, fangen wir mit einem echten *Tutorial* an: wir demonstrieren die Befehle des Debuggers zuerst bei der Lokalisierung eines recht einfachen Fehlers und kommen dann im weiteren Verlauf auf komplexere Funktionen zu sprechen.

Einige Worte zum Funktionsprinzip

Je nachdem, auf welcher »Ebene« eine Fehlersuche stattfindet, teilt man auch Debugger in verschiedene Klassen ein:

Maschinensprache-Debugger ermöglichen die schrittweise Ausführung einzelner Prozessorbefehle sowie die Ausgabe und Veränderung von Registerinhalten bzw. Speicherbereichen. Ein typischer Vertreter dieser Klasse ist das DOS-Programm DEBUG.

Symbolische Debugger arbeiten ebenfalls noch auf der Ebene der Maschinensprache, können aber mit Symbolen des untersuchten Programms umgehen (hauptsächlich: Variablen- und Routinennamen), wenn man ihnen die entsprechenden Informationen zur Verfügung stellt.

Quelltext-Debugger bewegen sich auf der Ebene des Quelltextes: Hier muß sich der Programmierer nicht mehr um einzelne Registerinhalte des Prozessors kümmern – stattdessen arbeitet er mit der Sprache, in der er das Programm auch erstellt hat.

Der integrierte Debugger von Turbo Pascal gehört der dritten Klasse an. Er wird mit Pascal-Befehlen und der Syntax von Pascal kontrolliert – so läßt sich

beispielsweise der Wert eines bestimmten Array-Elements mit einem Ausdruck wie

```
RecPtr^.Data[x+$80]
```

ausgeben und/oder verändern.

... und eine kurze Übersicht der Möglichkeiten

Zum Start des Debuggers sind keine besonderen Schritte notwendig – solange Sie mit den Standardvorgaben des Compilers arbeiten, wird der Debugger durch die Auswahl von *Run/Run* oder den Tastenbefehl Ctrl F9 automatisch aktiviert. Etwas detaillierter: *Options/Compiler/Debug information*, *Options/Compiler/Local symbols* und *Debug/ Integrated debugging* müssen sämtlich auf *On* gesetzt sein. (Was im übrigen auch der Standardvorgabe entspricht – bei allen Beispielen der vorangegangenen Kapitel war der Debugger also sozusagen unsichtbar dabei).

Durch das Setzen von *Breakpoints* läßt sich festlegen, an welchen Stellen die Ausführung des Programms angehalten werden soll. Wenn das Programm einen solchen Breakpoint erreicht, übergibt der Debugger die Kontrolle wieder an den Benutzer und stellt (unter anderem) die folgenden Möglichkeiten zur Verfügung:

- Ausgabe von Variablenwerten und/oder Ausdrücken, wobei sich das Format durch zusätzliche Angaben festlegen läßt;
- Definition von Ausdrücken, die Variablenwerte des Programms enthalten. Die Werte dieser Ausdrücke werden in einem eigenen Fenster ausgegeben und nach jedem Programmschritt auf den neuesten Stand gebracht;
- Veränderung von Variablenwerten;
- Setzen, Suchen und Löschen von Breakpoints;
- schrittweise Ausführung (jeweils eine Quelltextzeile pro Tastendruck);
- weitere Ausführung bis zum nächsten Breakpoint.

Außerdem stehen Ihnen natürlich sämtliche Möglichkeiten des Menüsystems von Turbo Pascal zur Verfügung – von der Veränderung des Quelltextes über eine erneute Compilierung bis hin zur Ausgabe von Verzeichnissen oder dem Aufruf der DOS-Kommandoebene.

Ein Beispiel

Drei der sieben Menüs von Turbo Pascal mit insgesamt knapp einem Dutzend Wahlpunkten sind ausschließlich der Fehlersuche gewidmet; dazu kommen Syntaxregeln, entsprechende Tastenkürzel und Fenster... Deshalb fangen wir erst einmal mit einem einfachen Beispiel an (und demonstrieren dabei, daß

auch der Debugger intuitiv beherrschbar ist): neun Zehntel der Möglichkeiten stehen dann zur Verfügung, wenn man sie braucht – und erledigen sich ansonsten durch die Standardvorgaben automatisch).

Unser Beispielprogramm hat den Namen *IndexFault* – und wie man sich in diesem Kontext wohl denken kann, enthält es einen recht ärgerlichen Fehler:

```
programIndexFault;   { Datei IDXFAULT.PAS }
{$D+,L+,R-}  { Debug information und Local symbols auf ON, keine
               Prüfung von Bereichen }
const
  CRLF = #13#10;
var
  List: array[1..10] of Integer;   { Dieses Array wird bearbeitet }
  Index: Integer;                  { Index in Liste }
```

Die Compiler-Befehle {$D+} und {$L+} sind genauso wie {$R-} eigentlich nicht notwendig, weil diese Modi den Standardvorgaben der integrierten Entwicklungsumgebung entsprechen.

```
procedure PrintList(Msg: string);
varx: Integer;
begin
  Write(Msg);
  for x:= 1 to 10 do   { Ausgabe der Array-Elemente }
    Write(List[x]:4);
  Writeln;
end;
```

PrintList gibt zuerst den als *Msg* übergebenen Text aus, darauf folgen die Elemente des Arrays *List* und schließlich ein Zeilenvorschub. Diese Routine wird sowohl nach der Initialisierung von *List* als auch nach dem Ende der Bearbeitung des Arrays aufgerufen.

```
begin         { Hauptprogramm }
  Writeln('VORSICHT: IDXFAULT ist ein Übungsprogramm für den', CRLF,
    'Debugger von Turbo Pascal. Wenn Sie dieses Programm', CRLF,
    '*nicht* von TURBO.EXE aus gestartet haben, dann drücken', CRLF,
    'Sie bitte *jetzt* Ctrl-Break - ansonsten hängt sich das', CRLF,
    'System auf!');
  Readln;
```

Das Hauptprogramm beginnt mit einer Warnung für ahnungslose Benutzer und der Möglichkeit zum Abbruch.

```
  for Index:= 1 to 10 do     { Initialisieren des Arrays }
    List[Index] := Index;

  PrintList('Liste ist initialisiert: ');   { Ausgabe }
```

Diese Schleife initialisiert die Elemente von *List* mit aufsteigenden Werten und gibt dann (via *PrintList*) sämtliche Elemente des Arrays aus.

```
  Index := 0;
```

```
  while (Index < 11) do      { Negation sämtlicher Elemente }
  begin
    Inc(Index,1);
    if List[Index] > 0 then
      List[Index] := - List[Index];
  end;

  PrintList('Ausgabe der negierten Werte: ');
end.
```

Die zweite Schleife des Hauptprogramms negiert sämtliche Elemente von *List* und ist zur Abwechslung nicht mit **for** aufgebaut. Nach der Ausgabe des veränderten Arrays (wiederum via *PrintList*) ist unser Beispielprogramm bereits beendet.

Der erste Start von IndexFault

Nachdem Sie IDXFAULT.PAS exakt in der gezeigten Form eingegeben haben, müssen Sie nur noch sicherstellen, daß der Schalter *Integrated Debugging* im Menü *Options/Debug* gesetzt ist (was ebenfalls eine Standardvorgabe darstellt) – danach können Sie das Programm via *Compile/Make EXE file* (oder durch einen Druck auf F9) compilieren lassen.

Wenn Compiler und Linker ihre Arbeit beendet haben, wählen Sie *Run/Step over* (oder benutzen den Tastenbefehl F8), um den Debugger zu starten. Als Ergebnis dieses Befehls stellt der Debugger den Teil des Quelltextes im Fenster *Edit* dar, der das Hauptprogramm enthält, und kennzeichnet das **begin** des Hauptprogramms mit einem hervorgehobenen Balken: diese Pascal-Anweisung wird als nächster Schritt ausgeführt, wenn wir das Programm fortsetzen.

Etwas Terminologie zwischendurch: Wir werden den Punkt, den der Debugger als nächsten auszuführenden Befehl kennzeichnet, im weiteren als *Startposition* bezeichnen; der Balken, der diesen Startpunkt kennzeichnet, bekommt konsequenterweise den Namen *Startbalken*.

Setzen Sie das Programm nun mit *Run/Run* (oder dem Tastenbefehl Ctrl F9) fort: Turbo Pascal schaltet auf den »DOS-Bildschirm« um, *IndexFault* gibt die Warnung für Ahnungslose aus und wartet dann auf die Eingabe von Enter. Nach der nun folgenden Ausgabe des initialisierten Arrays *List* herrscht Funkstille – wir hatten ja bereits gesagt, daß das Programm nicht so ganz fehlerfrei ist.

Drücken Sie Ctrl Break: Der Debugger schaltet auf den Bildschirm von Turbo Pascal zurück und fordert (über ein kleines Fenster) einen Druck auf Esc als Bestätigung dieses Abbruchs an.

Geben Sie nun den Tastenbefehl Ctrl F2 ein, um dem Debugger das Ende dieses Versuchs mitzuteilen, und starten Sie die Fehlersuche mit einem Druck auf F8 erneut. (Der Startbalken sollte danach wieder auf dem **begin** des Hauptprogramms stehen).

Ctrl Break

ist die einfachste Möglichkeit, ein außer Rand und Band geratenes Programm zu stoppen – vorausgesetzt allerdings, daß

- das Programm mit *Options/Debugger/Integrated debugging* und *Options/Com-*

piler/Debug information compiliert wurde (was beides den Standardvorgaben entspricht);

- Sie es von der integrierten Entwicklungsumgebung aus gestartet haben (also letztendlich über den Debugger, der – wie gesagt – auch bei »normalen« Starts immer unsichtbar dabei ist);

- und daß das Programm nicht Teile des Betriebssystems oder Teile von Turbo Pascal überschrieben oder auf andere Weise völlig durcheinander gebracht hat.

Wenn diese Voraussetzungen erfüllt sind, dann bricht die Eingabe von Ctrl Break das zu untersuchende Programm mit sofortiger Wirkung ab: der Debugger schaltet wieder auf den Bildschirm von Turbo Pascal um und setzt den Startbalken auf den Pascal-Befehl, der ohne die Unterbrechung als nächstes ausgeführt worden wäre.

Zum technischen Hintergrund: Der Debugger wird über den durch Ctrl Break ausgelösten Interrupt aufgerufen, lokalisiert die Startadresse des nächsten Pascal-Befehls und setzt dort einen Breakpoint (siehe nächster Abschnitt); danach wird das Programm erst einmal bis zu diesem Punkt fortgesetzt. Damit stellt der Debugger sicher, daß der Abbruch nicht inmitten einer Routine des Betriebssystems erfolgt.

Wenn Sie ein zweites Ctrl Break eingeben, *bevor* der Debugger das Programm endgültig gestoppt hat (technisch: einen weiteren Interrupt via Ctrl Break auslösen, während das Programm noch mit der Ausführung einer Routine des Betriebssystems beschäftigt ist), dann erzeugt der Debugger einen »echten« (sofortigen) Abbruch und unternimmt auch keinen weiteren Versuch, die entsprechende Stelle im Quelltext zu lokalisieren. Außerdem werden in diesem Fall weder Puffer entleert noch andere Maßnahmen für einen »ordentlichen« Abschluß ausgeführt. Anders gesagt: Sie sollten Ctrl Break wirklich nur dann ein zweites Mal eingeben, wenn Ihr Programm in einer endlosen Schleife hängengeblieben ist und durch das erste Ctrl Break aus irgendeinem Grund auch nach ein paar Sekunden noch nicht beendet wird.

Breakpoints

Wir hatten festgestellt, daß *IndexFault* das initialisierte Array *List* auf den Bildschirm ausgibt – was die Vermutung erlaubt, daß zumindest dieser Teil des Programms (also die **for**-Schleife des Hauptprogramms und die Routine *PrintList*) im erwünschten Sinne zu funktionieren scheint.

Aber Halt! Tatsächlich wurde nur bewiesen, daß die *Ausgabe* durch *PrintList* ordentlich funktioniert – ob das Programm jemals aus dieser Routine »zurückkommt«, wissen wir noch nicht. Setzen Sie den Cursor im Quelltext auf den ersten Aufruf von *PrintList*, also auf die Zeile

```
PrintList('Liste ist initialisiert: ');   { Ausgabe }
```

und wählen Sie danach den Punkt *Toggle breakpoint* im Menü *Debug* bzw. geben Sie Ctrl F8 ein. Der Debugger stellt nun diese Zeile hervorgehoben dar: Sie haben soeben festgelegt, daß das Programm nur bis zu dieser Stelle ausgeführt und dann angehalten werden soll, d.h. einen Breakpoint gesetzt.

Der Debugger verwendet unterschiedliche Farben bzw. Zeichenattribute für

den Startbalken und für Breakpoints. Der Startbalken sollte sich zu diesem Zeitpunkt nach wie vor auf dem **begin** des Hauptprogramms befinden – Cursorbewegungen im Editor sind jederzeit möglich und haben nichts mit der Ausführung von Pascal-Befehlen zu tun (die den Startbalken bewegen würden).

Wenn Sie nun *Run/Run* wählen (oder Ctrl F9 eingeben), dann wird die Ausführung von *IndexFault* ab der Position des Startbalkens fortgesetzt (was für unser Beispiel bedeutet, daß das Programm vom ersten Befehl an ausgeführt wird). Unterschiede zu einem Programmstart ohne gesetzte Breakpoints gibt es offensichtlich keine: Turbo Pascal schaltet wieder auf den »DOS-Bildschirm« um und zeigt Ihnen die Warnung des Programms. Geben Sie erneut Enter ein: *IndexFault* wird nun bis zu dem zuvor festgelegten Breakpoint fortgesetzt – danach hält der Debugger das Programm an, schaltet in das Fenster *Edit* zurück und zeigt nun den Startbalken auf dem Aufruf von *PrintList*. (Der hier gesetzte Breakpoint existiert nach wie vor – nur wird er im Moment durch den Startbalken verdeckt).

Schrittweises Ausführen von Befehlen: »Trace into« und »Step over«

Zur Ausführung dieses Befehls sollten Sie jetzt *nicht Run/Step over* bzw. F8 verwenden, sondern *Run/Trace into* bzw. F7. Der Debugger führt *PrintList* daraufhin nicht *en bloc* aus, sondern hält beim **begin** dieser Prozedur an.

Geben Sie den Befehl F7 erneut: Turbo Pascal führt das **begin** von *PrintList* (d.h. das Belegen von Speicherplatz für die lokale Variable *x*) aus und setzt den Startbalken auf den Aufruf von *Write*. Der Versuch, diesen Aufruf mit einem weiteren Druck auf F7 ebenfalls schrittweise ausführen zu lassen, bringt uns zu den Voraussetzungen für *Trace into*:

- die Routine (bzw. das Modul des Programms, das sie enthält), muß mit denselben Schaltern compiliert worden sein, die auch im aufrufenden Programm für die Arbeit mit dem Debugger notwendig sind (also *Options/Compiler/Debug information* und *Options/Compiler/Local symbols* gesetzt);
- der Quelltext dieses Moduls muß verfügbar sein.

Wenn diese Voraussetzungen nicht erfüllt sind, verhält sich *Trace into* genauso wie *Step over* – die Routine wird in einem Schritt ausgeführt. (Woraus auch folgt, daß sich Bibliotheksroutinen nie schrittweise verfolgen lassen – zum einen sind die Bibliotheken von Turbo Pascal mit *Options/Compiler/Debug information..Off* compiliert worden, zum anderen ist der dazugehörige Quelltext nicht verfügbar).

Ob der Aufruf von *Write* den gewünschten Effekt hatte, läßt sich übrigens jederzeit mit *Window/User screen* (oder Alt F5) nachprüfen: dieser Befehl schaltet auch bei der Arbeit mit dem Debugger auf den »DOS-Bildschirm« um. Wie sich zeigt, hatte *Write* das korrekte Ergebnis – neben dem Text

```
Liste ist initialisiert:
```

auf dem »DOS-Bildschirm« blinkt der Cursor und zeigt so die Position folgender Ausgaben an. Mit einem beliebigen Tastendruck gelangen Sie zum Bildschirm der integrierten Entwicklungsumgebung zurück.

»Go to cursor« – Ausführung bis zu einem bestimmten Punkt

Theoretisch ließe sich die **for**-Schleife in *PrintList* nun mit F7 oder F8 Schritt für Schritt verfolgen (wobei man nach jedem Aufruf von *Write* durch eine Umschaltung auf den »DOS-Bildschirm« prüfen könnte, ob er das gewünschte Ergebnis gezeitigt hat). Praktisch wäre so etwas recht mühsam – außerdem wissen wir bereits, daß die Ausgabe der Array-Elemente korrekt arbeitet: was wir prüfen wollten, war der Rücksprung zum Hauptprogramm.

Die naheliegende Idee wäre, mit Ctrl F8 auf dem abschließenden **end** von *PrintList* einen Breakpoint zu setzen und das Programm dann mit *Run/Run* bis zu diesem Punkt auszuführen – tatsächlich geht das aber einfacher:

Setzen Sie den Cursor auf das **end** von *PrintList* und wählen Sie *Run/Go to Cursor* (oder geben Sie den Tastenbefehl F4 ein). Der Debugger führt nun drei Schritte hintereinander aus:

- Definition eines Breakpoints auf der Cursorposition;
- Start bzw. Fortsetzen des Programms mit *Run/Run*;
- Löschen des Breakpoints.

Etwas kürzer: *Run/Go to cursor* bzw. F4 setzt einen *temporären* Breakpoint auf der Position des Cursors und führt das Programm bis zu diesem Punkt aus. Die Voraussetzungen dafür sind:

- Der Cursor muß sich auf einer Zeile befinden, die einen ausführbaren Befehl enthält. (Wenn er zwischen zwei Befehlszeilen steht – beispielsweise auf einem Kommentar – dann setzt der Debugger den Breakpoint auf den nächsten Befehl im Quelltext; falls sich der Cursor außerhalb aller Prozeduren und Funktionen befindet – beispielsweise in der **var**-Sektion – dann erhalten Sie eine Fehlermeldung des Debuggers, die mit Esc bestätigt werden muß).
- Das Programm muß den Befehl, auf dem der temporäre Breakpoint gesetzt wurde, auch tatsächlich ausführen – falls nicht (weil Sie den Cursor beispielsweise auf den **else**-Zweig einer Prüfung gesetzt haben, der nie ausgeführt wird), dann setzt der Debugger das Programm bis zu seinem Ende (oder einem permanenten Breakpoint) fort.

Hinweis: Wenn sich vor dem temporären Breakpoint ein (mit Ctrl F8 gesetzter) permanenter Breakpoint befindet, hält der Debugger das Programm dort an – nötigenfalls müssen Sie den F4 -Befehl entsprechend oft wiederholen.

Zurück zum Programm

Kommen wir zu *IndexFault* zurück: Der Startbalken sollte sich momentan auf dem **end** von *PrintList* befinden. Überprüfen Sie die Ausgaben der Routine mit einer Umschaltung auf den »DOS-Bildschirm« (via Alt F5), und geben Sie dann erneut den Befehl F7 . Der Debugger zeigt nun wieder das Hauptprogramm im Fenster *Edit* – wie es scheint, ist auch der Rücksprung von *PrintList* ohne Probleme ausgeführt worden.

Ob Sie den nächsten Schritt (die Zuweisung `Index := 0;`) mit *Trace into* oder *Step over* ausführen lassen, spielt keine Rolle (weil hier keine Aufrufe von Unterprogrammen mit im Spiel sind).

Mit dem Anfang der **while**-Schleife haben wir nun einen Punkt erreicht, ab dem der Fehler wahrscheinlich zu suchen ist – deshalb geht es hier Schritt für Schritt weiter. Wählen Sie *Run/Step over* (oder drücken Sie F8): Der Startbalken überspringt das **begin** der Schleife und steht nun auf dem Befehl:

```
Inc(Index, 1);
```

Lassen Sie auch diesen Befehl mit einem Druck auf F8 ausführen. Wie zu sehen, schaltet der Debugger dabei den Bildschirm nicht um: *Inc* ist zwar von der Syntax her ein Prozedur-Aufruf, wird aber vom Compiler mit direkten Befehlen codiert. (Dasselbe gilt übrigens für *Ord* und *Chr*, *Pred* und *Succ*, *Hi* und *Lo* – kurz: für alle Standardfunktionen, die auch innerhalb von berechneten Konstanten zulässig sind).

Ob der »Aufruf« von *Inc* den gewünschten Effekt hatte (nämlich die Erhöhung von *Index* auf den Wert 1), können wir jederzeit nachprüfen:

Ausgabe von Variablenwerten

Die Prüfung von Variablenwerten eines Pascal-Programms ist normalerweise eine recht umständliche Angelegenheit: Man füge an der entsprechenden Stelle einen Aufruf von *Writeln* ein, compiliere das Programm neu, starte es und sehe zu, was dabei herauskommt. Wenn das Ergebnis den Erwartungen entspricht, entferne man den *Writeln*-Befehl wieder und compiliere erneut...

Mit dem Debugger von Turbo Pascal geht das etwas einfacher. Die Auswahl von *Debug/Evaluate* erzeugt ein Fenster mit drei Feldern auf dem Bildschirm.

- In das Feld *Evaluate* wird der Variablenname (oder Ausdruck) eingegeben, dessen Wert der Debugger bestimmen soll. Wenn sich der Cursor auf einem Wort oder Variablennamen im Quelltext befindet, gibt Turbo Pascal diesen Text vor. Mit der Rechtspfeiltaste können Sie diese Vorgabe erweitern (d.h. zusätzliche Zeichen ab der Cursorposition kopieren – wir kommen später noch einmal darauf zurück).

- Im Feld *Result* gibt der Debugger den Wert der Variablen bzw. des Ausdrucks aus. Das Format der Ausgabe hängt vom Typ des Ausdrucks ab – über zusätzliche Angaben im Feld *Evaluate* läßt es sich auch explizit festlegen.

- Über eine Eingabe in das Feld *New value* läßt sich der im Feld *Evaluate* angegebenen Variablen ein neuer Wert zuweisen.

Mit den Aufwärts- und Abwärtspfeiltasten können Sie sich zwischen den drei Feldern bewegen; Ausdrücke und Werte, die sich nicht auf einmal in einem Feld darstellen lassen, werden mit der Links- und Rechtspfeiltaste sowie Home und End horizontal gerollt.

Ignorieren Sie eine eventuelle Vorgabe von Turbo Pascal im Feld *Evaluate* und geben Sie hier einfach *Index* ein. Nach Abschluß der Eingabe mit Enter zeigt das Feld *Result* das Ergebnis: wie zu erwarten, hat *Index* den Wert 1.

Wie sieht es mit den Werten von *List* aus? Sie lassen sich genauso einfach überprüfen: der Debugger stellt bei der Eingabe des Bezeichners *List* im Feld

Evaluate von selbst fest, daß es sich dabei um ein Array mit Integer-Elementen handelt. Dementsprechende Ausgaben erhalten Sie im Feld *Result*. Mit Alt E (oder Esc) gelangen Sie zum Editor zurück.

Die nächsten Befehle von *IndexFault* sollten nun die Vorzeichen der einzelnen Elemente von *List* umdrehen. Erfreulicherweise können wir uns die schrittweise Ausführung via *Run/Trace into* bzw. F7 sparen: Setzen Sie den Cursor auf den Anfang der **while**-Schleife, d.h. auf die Zeile

```
while (Index < 11) do      { Negation sämtlicher Elemente }
```

und wählen Sie *Toggle Breakpoint* aus dem Menü *Debug* bzw. geben Sie den Tastenbefehl Ctrl F8 ein. Die Auswahl von *Run/Run* (oder der Tastenbefehl Ctrl F9) führt das Programm nun bis zum nächsten Breakpoint aus – und da wir auf dem **while**-Befehl einen Breakpoint gesetzt haben, hält der Debugger *IndexFault* vor jedem weiteren Durchlauf der Schleife wieder an.

Nachdem man das einige Male ausprobiert und die Werte von *Index* bzw. *Liste* jedesmal erneut mit *Debug/Evaluate* überprüft hat, bleibt eigentlich nur ein schaudernder Ausblick: Was, wenn *List* 20, 30 oder 100 Elemente hätte?

Watch-Ausdrücke

Tatsächlich hört sich das nach wesentlich mehr Arbeit an, als es letztendlich ist: den weitaus größten Teil – nämlich die Überwachung der Variablenwerte – kann der Debugger übernehmen, wenn wir entsprechende *Watch*-Ausdrücke definieren. Worum es sich dabei handelt? Um Ausdrücke, die genauso formuliert und eingegeben werden wie bei einer Auswertung durch *Evaluate* – nur daß sie von Turbo Pascal im Fenster *Watch* permanent dargestellt und bei jedem Stop des Programms automatisch auf den neuesten Stand gebracht werden.

Das Fenster *Watch* belegt den unteren Teil des Bildschirms (was voraussetzt, daß *Edit* nicht auf die volle Bildschirmgröße gesetzt ist – gegebenenfalls müssen Sie mit F5 auf den »zweigeteilten« Bildschirm umschalten). Solange keine *Watch*-Ausdrücke definiert sind, ist *Watch* nur eine Bildschirmzeile hoch und leer – für jeden neu eingegebenen *Watch*-Ausdruck wächst es um eine Zeile, bis es die vorgegebene Größe des Fensters *Output* erreicht.

Definieren wir also zwei *Watch*-Ausdrücke für *Index* und *List*: Setzen Sie den Cursor im Quelltext auf ein beliebiges Vorkommen des Wortes *List* und wählen Sie entweder *Add watch* aus dem Menü *Debug/Watches* oder drücken Sie Ctrl F7 . Danach erscheint das Eingabefenster für *Watch*-Ausdrücke.

Wie zu sehen, gehen die Ähnlichkeiten mit *Evaluate* noch ein Stück weiter: Wenn der Cursor im Quelltext am Anfang oder innerhalb eines Wortes steht, gibt Turbo Pascal auch hier dieses Wort vor. (Falls Sie den Cursor vor der Aktivierung von *Add watch* nicht auf *List* gesetzt haben: wenn Sie als erstes ein »normales« Zeichen (d.h. keinen Befehl zu Steuerung des Cursors oder Enter) eingeben, wird eine eventuelle Vorgabe automatisch gelöscht).

Nach Abschluß der Eingabe mit Enter erscheint der Bezeichner *List* zusammen mit seinem momentanen Wert im Fenster *Watch*. Wiederholen Sie den

Tastenbefehl Ctrl F7 und geben Sie nun bitte nicht *Index* ein, sondern *Andechs*; Sie nehmen uns sonst die Möglichkeit zu einer weiteren Demonstration.

Zwei Dinge sollten hier deutlich werden: zum einen ist das Fenster *Watch* nach der Eingabe des zweiten Ausdrucks automatisch entsprechend vergrößert worden und umfaßt nun zwei Bildschirmzeilen; zum anderen kann Turbo Pascal mit der Angabe *Andechs* (oder was auch immer Sie getippt haben) nicht allzuviel anfangen. (Wieso der Ausdruck dann überhaupt akzeptiert wird? Weil das Fenster *Watch* auch Angaben aufnehmen kann, die momentan tatsächlich unbekannt sind – wie beispielsweise die Namen lokaler Variablen einer inaktiven Routine. Wir kommen später noch ausführlich darauf zurück).

Die bewußte Fehleingabe für *Index* bringt uns zu einem längeren Ausflug, bevor wir uns um das weitere Wohl und Wehe von *IndexFault* kümmern:

Hinzufügen, Bearbeiten und Löschen von Watch-Ausdrücken

Selbstverständlich kann man dem Fenster *Watch* nicht nur beliebige weitere Ausdrücke hinzufügen – einmal eingegebene Ausdrücke lassen sich auch bearbeiten und/oder wieder löschen.

Operationen mit *Watch*-Ausdrücken sind am einfachsten, wenn *Watch* das aktive Fenster ist: Verwenden Sie also den Tastenbefehl F6, um von *Edit* aus in dieses Fenster umzuschalten.

Turbo Pascal verwendet im Fenster *Watch* einen hervorgehobenen Balken als Cursor, der immer einen kompletten *Watch*-Ausdruck kennzeichnet und sich mit PgUp, PgDn, ← oder → bewegen läßt.

Fangen wir mit dem Bearbeiten eines Ausdrucks an: Setzen Sie den Cursor auf den Ausdruck *Andechs* und drücken Sie Enter. Daraufhin wird ein kleines Eingabefenster geöffnet, das diesen Ausdruck enthält und dieselben Tastenbefehle wie der Editor für Veränderungen zur Verfügung stellt.

Ändern Sie den Ausdruck von *Andechs* auf *Index*, und drücken sie Enter: Turbo Pascal ersetzt den alten Ausdruck im Fenster *Watch* durch die neue Eingabe und gibt nun auch den momentanen Wert von *Index* aus. Setzen Sie den Cursor auf *List*, drücken Sie auf Ins, und geben Sie den Ausdruck *Index**2 ein. Beachten Sie, wie das Ergebnis dieser Operation aussieht.

Womit wir zwei Dinge demonstriert hätten: Zum einen, daß *Watch*-Ausdrücke nicht auf einfache Variablennamen beschränkt sind (tatsächlich ist hier ein beliebiger Pascal-Ausdruck möglich, solange er keine Funktionsaufrufe enthält – wir kommen später noch ausführlich darauf zurück); zum anderen, daß Ins der Auswahl von *Add watch* im Menü *Debug/Watches* entspricht, solange *Watch* das aktive Fenster ist.

Der Cursor sollte momentan noch auf dem Ausdruck *Index**2 stehen – und dieser Ausdruck wurde tatsächlich nur zur Demonstration gebraucht: Wählen Sie entweder *Delete watch* aus dem Menü *Debug/Watches* oder drücken Sie auf Del (alternativ: Ctrl Y) – woraufhin Turbo Pascal diesen Ausdruck komplett löscht, das Fenster *Watch* wieder um eine Zeile verkleinert und den Cursor auf den nächsten *Watch*-Ausdruck (*List*) setzt.

Wenn Sie nun mit F6 in *Edit* hinüberwechseln, erscheint links neben *List* ein kleines rautenförmiges Symbol. Es kennzeichnet den »momentanen« Watch-Ausdruck – also die Stelle, an der sich der Cursor im Fenster *Watch* vor dem Wechsel befunden hat.

Mit der Auswahl von *Remove all watches* aus dem Menü *Debug/Watches* lassen sich übrigens sämtliche *Watch*-Ausdrücke auf einmal löschen (wobei es keine Rolle spielt, ob das Fenster *Watch* zum Zeitpunkt dieses Befehls aktiv ist oder nicht).

Zurück zum Programm

(Falls Ihnen nach diesem langen Einschub über das Fenster *Watch* die Übersicht ein wenig verloren gegangen sein sollte: Wir haben einen Breakpoint auf den Beginn der **while**-Schleife des Hauptprogramms gesetzt (Seite 141) und – mit leichten Umwegen – zwei *Watch*-Ausdrücke für die Variablen *Index* bzw. *List* definiert.

Was uns bleibt, ist die Ausführung dieser Schleife bis zu ihrem Ende: Geben Sie also so oft den Tastenbefehl Ctrl F9 (= *Run/Run*), bis *Index* den Wert 10 erreicht hat – da wir einen *Watch*-Ausdruck für *List* definiert haben, läßt sich sehr schön verfolgen, wie die Elemente dieses Arrays nacheinander ein negatives Vorzeichen bekommen.

Wenn Sie nun ein weiteres Mal den Tastenbefehl Ctrl F9 geben, zeigt sich ein erstaunliches Ergebnis: Nicht nur, daß die Schleife erneut durchlaufen wird, obwohl sie eigentlich nach dem zehnten Element beendet sein sollte – wie das Fenster *Watch* zeigt, hat *Index* plötzlich den Wert -11. Offensichtlich haben wir den Fehler in *IndexFault* gefunden.

»...« – eine kurze Pause

Bevor wir uns nun auf die **while**-Schleife stürzen und dem in ihr versteckten Fehler den Garaus machen, sollten wir kurz überlegen, was genau eigentlich passiert ist. Wieso bekommt *Index* plötzlich den Wert -11? Und: ist das der Grund für das »Aufhängen« des Programms?

Tatsächlich hängen diese beiden Erscheinungen zusammen. Weil die Schleife *Index vor* dem Zugriff auf *Liste* erhöht, ist die Prüfung über **while** um eins zu hoch angesetzt:

```
while (Index < 11) do      { <- Wiederholung für Index = 10: }
begin
  Inc(Index,1);            { <- Index wird auf 11 erhöht }
  if List[Index] > 0 then
    List[Index] := ä List[Index];  { und überschreibt hier das }
end;                       { (nicht definierte) Element List[11] }
```

Die Erklärung, wieso *Index* durch den Zugriff auf *List*[*11*] den Wert -11 bekommt, ist etwas komplizierter: *IndexFault* deklariert zuerst die Variable *List* und dann die Variable *Index*; *Index* ist deshalb direkt oberhalb von *List* im Datensegment des Programms gespeichert und wird – wie soeben demonstriert – durch einen Zugriff »über die Arraygrenzen hinaus« unbeabsichtigt verändert.

Was dabei herauskommt, ist eine endlose Schleife: Weil *Index* immer wieder als »*List*[*11*]« auf einen negativen Wert gesetzt wird, läuft das Programm solange,

bis wir es mit Ctrl Break (oder mit massiveren Maßnahmen wie dem RESET-Knopf) abbrechen.

Zum Sinn dieser kurzen Abschweifung: Wir haben zwar einen Fehler gefunden – es hätte aber genausogut sein können, daß er überhaupt nichts mit dem beobachteten Effekt (sprich: dem »Aufhängen« von *IndexFault*) zu tun hat. In einem solchen Fall würde man das Programm erst einmal um einige Schritte fortsetzen und dabei prüfen, welche Probleme als nächstes auftauchen.

Wie die Dinge aber nun einmal sind, können wir uns auf die Korrektur dieses Fehlers konzentrieren – die Wahrscheinlichkeit, daß *IndexFault* danach funktioniert (oder sich zumindest etwas vernünftiger verhält), ist hoch genug.

Zusammenfassung I

Vor der eigentlichen Korrektur (und dem Einführen weiterer Möglichkeiten) fassen wir die auf den letzten Seiten besprochenen Einzelheiten noch einmal kurz zusammen. Sie haben bis jetzt die folgenden Schritte ausgeführt:

- Compilierung des Programms, wobei die Schalter *Options/Compiler/Debug information* und *Options/Compiler/Local symbols* gesetzt waren und Start von Programm bzw. Debugger via *Step over* (wobei *Options/Debugger/Integrated debugging* gesetzt sein muß).

- Verfolgen des Aufrufs von *PrintList* mit *Trace into*. *Run/Step over* (bzw. F8) und *Run/Trace into* (bzw. F7) unterscheiden sich in einem Punkt: *Step over* führt Funktionen, die durch einen Pascal-Befehl aufgerufen werden, in einem Schritt aus; *Trace into* hält dagegen auf dem ersten Befehl einer aufgerufenen Funktion an und ermöglicht so deren schrittweises Verfolgen. Wenn diese Funktion nicht mit den »Debug-Schaltern« compiliert wurde oder kein Quelltext verfügbar ist, arbeitet *Trace into* genauso wie *Step over* – die aufgerufene Funktion wird in einem Schritt ausgeführt.

- Prüfung der Ausgaben von *PrintList* durch Umschalten auf den »DOS-Bildschirm« mit Alt F5 .

- Fortsetzen des Programms bis zum Ende von *PrintList* mit *Run/Go to cursor*. Über diesen Menüpunkt (oder den Tastenbefehl F4) läßt sich ein temporärer Breakpoint setzen – der Debugger führt daraufhin das Programm von der alten Position des Startbalkens bis zu dem Punkt aus, auf dem sich der Cursor momentan im Quelltext befindet.

- Ausgabe verschiedener Variablenwerte des Programms über *Debug/Evaluate* (oder Ctrl F4).

- Positionierung des Cursors auf den Start der **while**-Schleife im Hauptprogramm und Setzen eines Breakpoints via Ctrl F8 (bzw. dem Wahlpunkt *Toggle breakpoint* im Menü *Debug*).

- Fortsetzen des Programms mit *Run/Run* (oder Ctrl F9) und Ausführung bis zu diesem Breakpoint.

- Definition zweier *Watch*-Ausdrücke (*Index* bzw. *List*) und »Verfolgen« dieser beiden Variablen während der weiteren Ausführung der **while**-Schleife.

Fehlerkorrekturen und Neustart

Nachdem wir die **while**-Schleife des Hauptprogramms nun als Ursache allen Übels ausgemacht haben, stehen Ihnen mindestens zwei Möglichkeiten zur Verfügung:

- Nach einer direkten Veränderung von *Index* über *Debug/Evaluate* ließe sich das Programm ohne eine erneute Compilierung fortsetzen. Die notwendigen Schritte dazu wären: Aktivierung von *Evaluate* via Ctrl F4 , Eingabe des Bezeichners *Index* in das Feld *Evaluate* und Eingabe des Wertes 11 in das Feld *New value*. Danach könnte man das Programm entweder in Einzelschritten oder (für wagemutige Optimisten) mit *Run/Run* zu Ende bringen.
- Korrektur des Quelltextes und erneute Compilierung.

Wir haben uns für die zweite Möglichkeit entschieden, weil sich damit einige weitere Eigenschaften des Debuggers demonstrieren lassen.

Beenden Sie den Debugger mit *Run/Program reset* (oder dem Tastenbefehl Ctrl F2 : Turbo Pascal gibt daraufhin den von *IndexFault* belegten Platz im Hauptspeicher frei, schließt offene Dateien (was in unserem Beispiel keine weiteren Auswirkungen hat) und löscht den Startbalken im Fenster *Edit*.

Eine »schöne« Korrektur der **while**-Schleife könnte in etwa so aussehen:

```
Index := 1;             { <- Startwert 1 - und nicht 0 }
while (Index < 11) do   { ... natürlich könnte man im Prinzip auch }
begin                   { einfach den Grenzwert "herunterdrücken" }
  if List[Index] > 0 then
    List[Index] := - List[Index];
  Inc(Index,1); { <- Erhöhung von Index NACH dem Zugriff auf List }
end;
```

Speichern Sie die neue Version von *IndexFault* und wählen Sie *Run/Run* (oder geben Sie den Tastenbefehl Ctrl F9) ein.

Turbo Pascal stellt fest, daß der Quelltext seit der letzten Compilierung verändert wurde, und fragt nach, ob der Quelltext neu compiliert werden soll. Bestätigen Sie diese Anfrage mit Y . Nachdem Compiler und Linker ihre Arbeit beendet haben, wird *IndexFault* erneut gestartet, meldet sich mit der (nun hoffentlich unnötigen) Warnung für Ahnungslose und wartet auf die Eingabe von Enter .

Weitere Details zu Breakpoints

Falls Sie Turbo Pascal seit der ersten Untersuchung des Programms mit dem integrierten Debugger nicht (mindestens) einmal vollständig verlassen haben, dann kommt *IndexFault* beim Start mit *Run/Run* nicht einmal bis zur Ausgabe des initialisierten Arrays – auf dem Befehl

```
PrintList('Liste ist initialisiert: ');   { Ausgabe }
```

ist nämlich nach wie vor ein Breakpoint gesetzt – Breakpoints bleiben offensichtlich auch zwischen zwei Starts des Debuggers erhalten.

Tatsächlich ist die Lebensdauer von Breakpoints fast unbegrenzt: Sie überstehen nicht nur das Beenden und Neustarten des Debuggers, sondern auch Veränderungen des Quelltextes und erneute Compilierungen (und sind dabei nicht an eine bestimmte Stelle des Quelltextes, sondern an den entsprechenden Befehl gebunden, verschieben sich beim Löschen oder Einfügen von Quelltext-Zeilen also automatisch). Damit nicht genug: Auch wenn Sie nun den Quelltext von *IndexFault* auf der Diskette speichern, einen anderen Text bearbeiten und schließlich IDXFAULT.PAS erneut laden – der Breakpoint ist nach wie vor präsent. Einmal gesetzte Breakpoints werden nur wieder gelöscht, wenn Sie

- den Cursor auf die entsprechende Zeile setzen und danach *Toggle breakpoint* aus dem Menü *Debug* wählen (oder den Tastenbefehl Ctrl F8 geben);
- sämtliche Breakpoints mit *Clear all* im Breakpoint-Dialog in *Debug/Breakpoints* zurücksetzen;
- die Quelltext-Zeile vollständig löschen, auf der der Breakpoint gesetzt ist;
- oder Turbo Pascal beenden.

Es gibt allerdings zwei Fälle, in denen Turbo Pascal gesetzte Breakpoints *verlieren* kann:

- Wenn Sie einen Quelltext bearbeiten, sich irgendwann entschließen, diese neue Version nicht zu speichern und statt dessen das Original wieder laden. Turbo Pascal paßt zuvor gesetzte Breakpoints den Veränderungen im neuen Text an, merkt sich aber nicht, wo sich diese Punkte in der alten Version der Datei befunden haben – und stellt sie beim erneuten Laden dieser alten Dateiversion deshalb auf falschen Positionen dar.

  Bevor Sie Veränderungen eines Quelltextes dadurch rückgängig machen, daß Sie die Originalversion dieser Datei erneut von der Diskette laden, sollten Sie deshalb sämtliche Breakpoints löschen.

  Achtung! Die Auswahl von *Clear all* im Breakpoint-Dialog setzt *sämtliche* Breakpoints zurück, die Turbo Pascal momentan gespeichert hat – die Löschung erstreckt sich also nicht nur auf den Quelltext im Editor, sondern auch auf alle anderen Dateien, zu denen momentan Breakpoints gespeichert sind.

- Wenn Sie einem Quelltext neue Zeilen hinzufügen oder Zeilen löschen, während der Debugger aktiv ist und die Arbeit mit dem Debugger dann fortsetzen, ohne das Programm neu zu compilieren. In diesem Fall sind die Breakpoints zwar nach wie vor an den richtigen Positionen – weil aber der Quelltext nicht mehr mit dem ausführbaren Programm übereinstimmt, werden sie an den falschen Stellen im Quelltext dargestellt. (Dasselbe gilt für den Startbalken, dessen Position im Quelltext dann auch nicht mehr unbedingt stimmen muß).

  Zufällig kann diese Situation allerdings kaum entstehen, weil Turbo Pascal nach Veränderungen des Quelltextes vor der nächsten Operation mit dem Debugger explizit zurückfragt, ob das Programm neu compiliert werden soll.

Vor der Compilierung eines Quelltextes lassen sich Breakpoints auch an Stellen setzen, die keine ausführbaren Pascal-Befehle enthalten (beispielsweise auf Kommentar- oder Leerzeilen). Derartige Punkte sind aber nicht sonderlich

sinnvoll (und zudem wirkungslos) – Turbo Pascal gibt deshalb eine Warnung aus, wenn Sie versuchen, das Programm zu einem späteren Zeitpunkt zu starten.

Wenn der Quelltext schon einmal compiliert und seit dieser Compilierung nicht weiter verändert wurde, dann weiß der Compiler, an welcher Stelle ausführbare Pascal-Befehle stehen und wo nicht – hier erhalten Sie sofort eine Fehlermeldung beim Versuch, einen Breakpoint an der falschen Stelle zu setzen.

Mit *View* aus dem Breakpoint-Dialog in *Debug/Breakpoints* läßt sich der Cursor übrigens auf den Breakpoint im Quelltext setzen – dieser Menüpunkt stellt eine recht einfache Möglichkeit dar, einzelne Breakpoints in einem Programm wiederzufinden (und/oder separat zu löschen).

Ein letztes Mal: Zurück zum Programm

Da *IndexFault* gleich zwei permanent gesetzte Breakpoints enthält, dürfte *Clear all* aus dem Menü *Debug/Breakpoints* die einfachere Möglichkeit sein, zu einem lauffähigen Programm zu kommen. (Wo der zweite Breakpoint war? Wie sich *vor* dem Löschen über *Clear all* mit *View* aus dem Menü *Debug/Breakpoints* herausfinden läßt – auf dem Beginn der **while**-Schleife im Hauptprogramm).

Danach sollte das Programm eigentlich tun, was man von ihm erwartet.

Zusammenfassung II

Vor den Listen und Tabellen des Referenzteils noch einmal ein kurzer Rückblick auf die besprochenen Befehle:

- der Debugger wird mit Ctrl F2 oder *Run/Program reset* beendet. Turbo Pascal gibt auf diesen Befehl hin den vom Programm belegten Speicherplatz frei und schließt alle offenen Dateien.

- Permanent gesetzte Breakpoints sind ausgesprochen langlebig und überstehen auch einen Wechsel des Quelltextes. Sie lassen sich entweder einzeln (durch Positionierung des Cursors und Ctrl F8) oder alle auf einmal löschen (mit *Clear all* aus dem Menü *Debug/Breakpoints*).

Fehlersuche in größeren Projekten

Sämtliche Befehle des Debuggers sind natürlich auch auf Programme anwendbar, die sich aus mehreren Units und/oder Overlays zusammensetzen. Hier kommen noch einige weitere Funktionen hinzu, die wir der Einfachheit halber anhand des Programms *IndexFault* demonstrieren:

Suche nach Prozeduren und Funktionen

Starten Sie *IndexFault* ein weiteres Mal über den Debugger (wobei die Ausführung des allerersten Schrittes via F8 ausreichend ist) und wählen Sie danach *Search/Find procedure*: Turbo Pascal öffnet nun ein Dialogfenster und erwartet die Eingabe eines Prozedur- oder Funktionsnamens. Geben Sie *PrintList* ein:

Wenn Sie diese Eingabe mit Enter abgeschlossen haben, zeigt der Editor den ersten ausführbaren Befehl dieser Routine (und den restlichen dazugehörigen Quelltext).

Dasselbe ließe sich in einem Programm wie *IndexFault*, das nur aus einem einzigen Quelltext besteht, auch über die Suchfunktionen des Editors erreichen – bei einem wirklich großen Programm, wo die Zahl der Module ohne weiteres bei mehreren Dutzend liegen kann, sieht es dagegen etwas anders aus. Im Gegensatz zum Editor findet *Find procedure* auch in solchen Fällen die richtige Quelltext-Datei (sofern sie verfügbar ist), lädt sie von der Diskette und stellt den ersten Befehl der gesuchten Routine im Editorfenster dar.

Sinnvoll ist *Find procedure* unter anderem dann, wenn Sie eine Routine verändern wollen (oder müssen) und nicht mehr genau wissen, in welchem Modul des Programms sie sich befindet.

Hinweis: Find procedure funktioniert auch dann, wenn der Debugger nicht aktiv ist – dieser Wahlpunkt setzt lediglich voraus, daß das entsprechende Programm mit *Debug information* compiliert wurde.

Ausgabe des »Return-Stacks«

Auch dieses Beispiel fällt in einem Programm wie *IndexFault* recht trivial aus: Setzen Sie den Cursor auf den ersten Befehl der Prozedur *PrintList* und verwenden Sie *Run/Go to cursor* (oder F4), um das Programm bis zu diesem Punkt ausführen zu lassen.

Die Auswahl von *Window/Call stack* (oder der Tastenbefehl Ctrl F3) bringt nun den momentanen Zustand des »Return-Stacks« in einem eigenen Fenster auf den Bildschirm.

In diesem »Stack« sind die Routinen verzeichnet, über die der momentane Punkt erreicht wurde. Das Hauptprogramm steht (mit dem als **program** angegebenen Namen) immer als unterstes, weil von dort aus sämtliche anderen Routinen aufgerufen wurden die Funktion oder Prozedur, in der das Programm gestoppt wurde, befindet sich als oberste auf dem Stack.

Neben jedem angegebenen Routinennamen sind die Werte der übergebenen Parameter verzeichnet – wenn sie sich aus Platzgründen nicht vollständig darstellen lassen, können die Links- und Rechtspfeiltaste sowie Home und End verwendet werden, um den entsprechenden Eintrag horizontal zu rollen.

Mit ← und → läßt sich der Cursor zwischen den Elementen dieses Stacks bewegen; ein Druck auf Enter bringt den Punkt im Quelltext auf den Bildschirm, von dem aus der Aufruf stattgefunden hat. Um beispielsweise festzustellen, von welchem Punkt innerhalb des Hauptprogramms die Routine *PrintList* aufgerufen wurde, setzen Sie den Cursor auf den Eintrag *INDEXFAULT* und drücken Enter.

Call stack kann in zwei Situationen eine unschätzbare Hilfe sein:

- wenn Sie einen Breakpoint in einer Routine gesetzt haben, die über zehn oder zwanzig Ebenen hinweg aufgerufen wurde, und nun herausfinden wollen, wie und von wo aus dieser Aufruf letztendlich geschah;

- wenn Sie nach längeren Suchoperationen im Quelltext (inklusive dem Wechseln von Dateien) zum Startbalken zurückkehren wollen: In diesem Fall setzen Sie einfach den Cursor auf den obersten Eintrag des »Return-Stacks« und drücken Enter.

Call stack kann (wie alle anderen Funktionen des Debuggers) mit einer fast beliebigen Zahl von Modulen eines Programms arbeiten – falls nötig, wird die entsprechende Quelltext-Datei automatisch in den Editor geladen.

Korrekturen in mehreren Modulen
sind weder für Turbo Pascal noch für den integrierten Debugger ein Problem: Gesetzte Breakpoints werden automatisch verschoben; vor dem Laden eines anderen Quelltextes erhalten Sie gegebenenfalls eine Rückfrage, ob die momentane (veränderte) Datei gespeichert werden soll.

Tatsächlich liegen die Probleme hier ausschließlich beim Programmierer: Je höher die Zahl der Dateien ist, die man in einem Korrekturgang verändert, desto größer ist auch die Wahrscheinlichkeit, daß sich dabei neue Fehler einschleichen – man sollte sich deshalb auch bei Korrekturen an größeren Projekten immer auf eine möglichst kleine Anzahl von Modulen beschränken. (Das setzt natürlich eine konsequente Abgeschlossenheit der einzelnen Module des Programms voraus und fällt somit in die Rubrik »defensive Programmierung« – ein Thema, auf das wir in diesem Kapitel noch einmal zurückkommen).

Teil II: Die Befehle des Debuggers

Dieser Referenzteil besteht aus zwei Abschnitten:

- einer tabellarischen Auflistung der Tastenbefehle;
- und einer detaillierten Besprechung der Syntax von Ausdrücken für *Watch* und *Evaluate* zusammen mit Formatierungsmöglichkeiten, »qualifizierten« Variablennamen und Vorgaben durch Turbo Pascal.

Tastenbefehle des Debuggers und verwandte Kommandos

Im ersten Teil dieser Besprechung – dem *Tutorial* – haben wir sämtliche Tastenbefehle des Debuggers in Rahmen der Beispiele vorgestellt. Hier sind sie noch einmal zusammen mit den wichtigsten Compiler-Schaltern in tabellarischer Form zusammengefaßt:

Tabelle 6.1 Tastenkürzel des Debuggers

Tastenkürzel, Menüpunkt	Beschreibung
Ctrl F9 *Run/Run*	prüft ein Programm via Compile/MAKE, startet es und führt es bis zum nächsten permanent gesetzten Breakpoint aus bzw. setzt die Programmausführung nach einer Unterbrechung fort.

Tastenkürzel, Menüpunkt	Beschreibung
Ctrl F2 *Run/Program reset*	beendet den Debugger, gibt belegten Speicherplatz frei, schließt alle offenen Dateien und hat nur eine Wirkung, wenn der Debugger zuvor aktiv war.
F4 *Run/Go to cursor*	setzt einen temporären Breakpoint auf der Cursorposition und startet das Programm bzw. setzt es ab der momentanen Position des Startbalkens fort. Falls noch nicht geschehen, wird der Debugger aktiviert.
F7 *Run/Trace into*	führt den Pascal-Befehl an der Position des Startbalkens aus. Wenn dieser Befehl den Aufruf einer Pascal-Routine enthält, die ebenfalls mit den »Debug-Schaltern« compiliert wurde, hält der Debugger auf dem ersten Befehl dieser Routine an; sonst wie *Step over*. Falls noch nicht geschehen, wird der Debugger aktiviert.
F8 *Run/Step over*	führt den Pascal-Befehl an der Position des Startbalkens aus und bearbeitet Aufrufe von Unterprogrammen in einem Schritt. Falls noch nicht geschehen, wird der Debugger aktiviert.
O/C/Debug information	muß bei der Compilierung auf *On* gesetzt sein, damit sich das Programm über den Debugger prüfen läßt.
O/C/Local symbols	muß bei der Compilierung auf *On* gesetzt sein, wenn lokale Variablen verfolgt werden sollen.
Ctrl F4 *Debug/Evaluate/modify*	berechnet den Wert eines Ausdrucks und erlaubt die Veränderung von Variablenwerten.
Ctrl F3 *Window/Call stack*	stellt den »Return-Stack« (d.h. die Reihenfolge aufgerufener Routinen) dar und ermöglicht das Ermitteln von Aufrufstellen. Nur verwendbar, wenn der Debugger aktiv ist.
Search/Find procedure	sucht eine Routine und stellt ihren Quelltext im Editor dar. Nur verwendbar, wenn das Programm vollständig compiliert wurde.
Options/Debugger/Integrated debugging	muß auf *On* gesetzt sein, damit sich ein Programm über den integrierten Debugger ausführen läßt.
Options/Debugger/ Standalone debugging	muß auf *On* gesetzt sein, wenn das Programm über den *Turbo Debugger* geprüft werden soll.
Options/Debugger/ Display swapping	legt fest, wann der Debugger auf den »DOS-Bildschirm« umschaltet.
System-Menü/ Refresh display	gibt den Bildschirm der integrierten Entwicklungsumgebung neu aus.

Tastenkürzel, Menüpunkt	Beschreibung
Ctrl F7 *Debug/Watches/ Add watch*	erwartet die Eingabe eines *Watch*-Ausdrucks und fügt diesen Ausdruck dem Fenster *Watch* hinzu.
Debug/Watches/ Delete watch	löscht den *Watch*-Ausdruck, auf dem sich der Cursor befindet.
Debug/Watches/ Remove all watches	löscht sämtliche *Watch*-Ausdrücke.
Debug/Watches/ Edit watch	ermöglicht die Veränderung des *Watch*-Ausdrucks, auf dem sich der Cursor befindet.
Ctrl F8 *Debug/Toggle breakpoint*	schaltet den Breakpoint an der Position des Cursors um, d.h. setzt/löscht einzelne Breakpoints.
Debug/Breakpoints/Clear all	löscht sämtliche Breakpoints, die Turbo Pascal momentan gespeichert hat.
Debug/Breakpoints/View	setzt den Cursor auf den Breakpoint im Quelltext.

Ausdrücke für Watch und Evaluate

Für *Watch*-Ausdrücke oder Eingaben im Feld *Evaluate* der Funktion *Debug/Evaluate* ist jeder Ausdruck zulässig, der der Syntax von Pascal entspricht, solange er nicht eines der beiden folgenden Elemente enthält:

- Funktionsaufrufe des laufenden Programms (die wegen möglicher Seiteneffekte grundsätzlich gesperrt sind). Die folgenden Standardfunktionen lassen sich dagegen frei verwenden:

Abs	*IOResult*	*Mem*	*Ofs*	*Round*	*SSeg*
Addr	*Length*	*MemAvail*	*Ord*	*Seg*	*Succ*
Chr	*Lo*	*MemL*	*Pred*	*SizeOf*	*Swap*
Hi	*MaxAvail*	*MemW*	*Ptr*	*SPtr*	*Trunc*

- lokale Variablen oder typisierte Konstanten, die außerhalb des Geltungsbereichs der Funktion liegen, in der der Debugger das Programm angehalten hat – es sei denn, die entsprechenden Bezeichner sind vollständig »qualifiziert« (siehe übernächster Abschnitt).

Ansonsten ist hier (inklusive expliziter Typ-Umwandlungen) praktisch alles erlaubt, was auch innerhalb eines Programms möglich ist:

```
(Abs(XVal) + Ord('A')) div 17
MaxAvail - SizeOf(MyRec) + 4 + Length(XStr)
Trunc(Real_Var)+Int_Var=Int_Array[x+$80] {<- ein boolscher Ausdruck}
MemW[$40,$17] + ^Word(Pointer(Ptr(SSeg, SPtr)))
```

```
Byte(XString[0])
FileRec(XFile)   { <- stellt die Dateivariable XFile als Record dar }
```

Der Debugger legt das Format der Ausgabe normalerweise über den Ergebnistyp des Ausdrucks fest – für einen Zeiger auf *Char* werden folglich Zeichen ausgegeben, für Variablen des Typs *Integer* ganzzahlige (dezimale) Werte usw.

Abweichend von der Pascal-Syntax ist zusätzlich noch die explizite Festlegung eines bestimmten Ausgabeformats erlaubt – auf die entsprechenden Möglichkeiten kommen wir in einem der folgenden Abschnitte noch zu sprechen.

Tabelle 6.2 gibt die verwendbaren Datentypen zusammen mit den dazugehörigen Ausgabeformaten wieder:

Tabelle 6.2 Datentypen und Ausgabeformate für Evaluate bzw. Watch

Typ	Erläuterung/Beispiel
Integer	dezimale Notation: 10, -23
Gleitkomma	die Ausgabe erfolgt ohne Exponenten, soweit möglich. Beispiele: 23476.374, 0.000234, 1.24e+104
Zeichen	»druckbare« Zeichen inklusive des erweiterten ASCII-Zeichensatzes werden in Hochkommas und direkt ausgegeben; Zeichen mit ASCII-Codes im Bereich von 0..31 in der Form #nn. Beispiele: 'A', 'ü', #4
Booleans	TRUE oder FALSE. Sowohl *Evaluate* als auch *Watch* stellen die Ergebnisse logischer Operationen beliebiger Art (wie beispielsweise @X =@XFunc.X, NPtr =Ptr($40,$12) oder StrVal[x]>#4) in boolscher Form dar
Aufzählungstypen	werden mit den entsprechenden Namen (in Großbuchstaben) ausgegeben. Beispiele: MONTAG, DIENSTAG, MITTWOCH, GREEN
Zeiger	die Ausgabe erfolgt in der Form PTR(Segment, Offset), wobei Segment und Offset hexadezimal notiert sind. Beispiele: PTR($40,$0), PTR($3EAF,$0C)
Strings	Ausgabe des momentanen Inhalts, eingeschlossen in Hochkommas. Beispiele: 'Rudi Ratlos', 'Schunck'
Arrays	Ausgabe in Klammern, wobei die einzelnen Elemente durch Kommas voneinander getrennt sind; das Format ergibt sich aus dem Elemententyp. Mehrdimensionale Arrays werden wie bei einer Deklaration als typisierte Konstanten dargestellt. Beispiele: (50,60,70), ((10,20),(30,40),(50,60))
Records	Ausgabe in Klammern, wobei die einzelnen Felder durch Kommas voneinander getrennt sind. Das Format der Felder ergibt sich durch den jeweiligen Feldtyp; mehrdimensionale Records werden wie bei einer Deklaration als typisierte Konstanten dargestellt. Beispiele: ('Fritz','Meier',100,ROT,TRUE), (5.6,7.8,(TRUE,'O',9), MONTAG)

Typ	Erläuterung/Beispiel
Mengen	Ausgabe in eckigen Klammern, wobei die einzelnen Elemente durch Kommas voneinander getrennt sind; soweit möglich, werden Unterbereiche gebildet. Beispiele: [MO,MI,FR], ['0'..'9','A'..'F']
Dateien	Ausgabe des momentanen Status, wobei zwischen den vier Zuständen CLOSED, OPEN, INPUT und OUTPUT unterschieden wird, sowie des zugeordneten Dateinamens als String. Beispiele: (OPEN, 'BUDGET.DTA'), (INPUT, 'MYDATA.TXT') Über eine explizite Typ-Umwandlung mit dem in der Unit *Dos* definierten Typ *FileRec* und zusätzliche Format-Angaben lassen sich die Felder einer Dateivariablen darstellen. Wenn *DVar* eine Dateivariable ist, dann ergibt der Ausdruck FileRec(DVar),RX eine Ausgabe wie (HANDLE:$6;MODE:$D7B3;RECSIZE:$14;PRIVATE:($0,$0,$0...))

Explizite Formate und Ausgaben

Wie bereits erwähnt, richten sich *Evaluate* und *Watch* bei der Darstellung eines Wertes normalerweise nach dem Ergebnistyp des Ausdrucks – ein Ausdruck des Typs *Integer* wird also als (dezimaler) Zahlenwert ausgegeben, ein Ausdruck des Typs *Char* als einzelnes Zeichen usw.

Dieser Mechanismus läßt sich durch eine explizite Formatangabe außer Kraft setzen, die auf den Ausdruck folgt und durch ein einzelnes Komma von ihm abgetrennt wird. Eine Formatangabe besteht aus einem optionalen *Wiederholungszähler*, auf den ein oder mehrere Buchstaben folgen, deren Interpretation in Tabelle 6.3 zusammengefaßt ist. Groß- und Kleinschreibung spielen keine Rolle, Leerzeichen zwischen dem Wiederholungszähler und darauffolgenden Buchstaben sind nicht notwendig.

Der Wiederholungszähler legt fest, wieviele aufeinanderfolgende Werte ausgegeben werden sollen, und wird normalerweise für Arrayelemente verwendet. Wenn *List* beispielsweise ein Array mit 10 Elementen darstellt, dann geben *Evaluate* und *Watch* den Wert des Ausdrucks *List* in der folgenden Form an:

```
List: (10, 20, 30, 40, 50, 60, 70, 80, 90, 100)
```

(Dieses Beispiel geht natürlich davon aus, daß die Elemente von *List* zuvor initialisiert wurden). Die folgende Formatangabe legt ein erstes Element fest und spezifiziert dann, wieviele folgende Elemente ausgegeben werden sollen:

```
List[5],3: (50, 60, 70)
```

Diese Begrenzung empfiehlt sich vor allem bei Datenstrukturen, die sich wegen ihres Umfangs nicht auf einer einzelnen Zeile darstellen lassen.

Wiederholungszähler sind nicht auf Arrays beschränkt – generell gilt, daß Turbo Pascal bei einer Angabe der Form *Variable*,*n* ab der durch *Variable* angegebenen Speicheradresse *n* aufeinanderfolgende Werte des Typs von *Variable* ausliest und darstellt. Hier gibt es aber eine Begrenzung: der angegebene Ausdruck muß tatsächlich einer Variablen entsprechen (d.h. auf der linken Seite einer Pascal-Zuweisung legal sein) – ansonsten wird der Wiederholungszähler ignoriert.

Tabelle 6.3 Formatangaben für Watch und Evaluate

Zeichen	Funktion/Wirkung
C	»character« – Steuerzeichen (ASCII-Codes 0 bis 31) werden im IBM-Bildschirmzeichensatz wiedergegeben. (Ein Zeichen mit dem ASCII-Code $03 erscheint z.B. als Spielkarte). Anwendbar auf Zeichen, Strings und Integer.
S	»string« – Steuerzeichen (ASCII-Codes 0 bis 31) werden in der Form #*nn* dargestellt. Da dieses Format der Standardvorgabe für einzelne Zeichen und Strings entspricht, ist die explizite Formatangabe »S« nur zusammen mit »M« sinnvoll.
D	»decimal« – Ganzzahlige Werte werden in dezimaler Form dargestellt. Anwendbar auf Integer-Ausdrücke, Arrays und Records mit Integer- oder Zeichen-Elementen.
H oder X	»hex« – Ganzzahlige Werte werden in hexadezimaler Form mit einem vorangestellten $ dargestellt. Anwendbar auf Integer-Ausdrücke, Zeichen, Arrays und Records mit Integer-Elementen.
F*n*	floating point – *n* steht für einen Integer im Bereich von 2 bis 18 und legt die Anzahl der Nachkommastellen bei der Ausgabe fest. Nur auf Gleitkommazahlen anwendbar.
M	»memory dump« – Speicherauszug ab der durch den Ausdruck angegebenen Adresse. Der Ausdruck muß auf der linken Seite einer Pascal-Zuweisung legal sein, d.h. eine Speicheradresse ergeben; ansonsten wird die Formatangabe »M« ignoriert. Standardmäßig werden die gelesenen Werte im Hexadezimalformat (ohne vorangestelltes »$«) ausgegeben. In Kombination mit »D« (*Ausdruck,MD*) gibt Turbo Pascal die gelesenen Werte in dezimaler Form aus, die Kombination von »M« mit »H« erzeugt hexadezimale Ausgaben mit vorangestelltem »$«. Bei Anfügen eines »C« oder »S« gibt Turbo Pascal die Werte in der Form »character« bzw. »string« aus, d.h. stellt ASCII-Zeichen dar. Die Anzahl der ausgegebenen Bytes wird standardmäßig durch die Größe der als *Ausdruck* angegebenen Variablen bestimmt, läßt sich durch einen zusätzlichen Wiederholungszähler aber ebenfalls explizit festlegen.
P	»pointer« – Ausgabe der Adresse, auf die eine Zeigervariable zeigt. Turbo Pascal verwendet hier nicht die Standardform *PTR*(*Seg*, *Ofs*), sondern *Segment:Offset*, d.h. trennt die beiden Werte durch einen Doppelpunkt. Nur auf Zeigervariablen anwendbar.
R	»record« – Ausgabe einzelner Felder, deren Typ durch den Ausdruck festgelegt wird (wie z.B. (*X: 1, Y: 10, Z : 'Hello'*)). Nur auf Records und Varianten anwendbar.

Es gelten drei generelle Regeln:

1. Formatangaben, die nicht auf den Typ des zuvor angegebenen Ausdrucks anwendbar sind, werden von Turbo Pascal ignoriert.

2. Wenn der auszugebende Ausdruck Elemente unterschiedlichen Typs enthält (d.h. eine Struktur oder Variante ist) und auf ihn mehr als eine Formatangabe folgt, dann wendet Turbo Pascal automatisch die jeweils »passenden« Formatangaben an. Bei einer Angabe wie *RecVar,F5H* werden folglich sämtliche Mitglieder von *RecVar*, die Gleitkommazahlen enthalten, mit fünf Stellen nach dem Komma ausgegeben (»F5«); Integerwerte stellt der Debugger in hexadezimaler Notation dar (»H«).

3. Bei mehr als einer Formatangabe gleichen Typs für einen Ausdruck wird automatisch das Format mit der höchsten Rangstufe verwendet (siehe »Operatoren« in Kapitel 6 »Ausdrücke« des *Programmierhandbuchs*). Diese Regelung ist im allgemeinen nur bei der Ausgabe von Records und Varianten von Belang – für einfache Variablen und Arrays, deren Elemente aus einfachen Variablen bestehen, wird man normalerweise nur ein Format festlegen.

Hinweis: Die Anwendung des Formatangabe »H« auf eine Zeigervariable hat die (hexadezimal dargestellte) Adresse zum Ergebnis, auf die diese Zeigervariable zeigt.

Einige Beispiele zu expliziten Formatangaben

Zur Demonstration definieren wir ein Programm wie

```
type
  NamePtr = ^NameRec;
  NameRec = record
      Next: NamePtr;
      Count: Integer;
      Name: string[31];
    end;

const List: array[1..5] of Integer = (10,20,30,40,50);
var   P: NamePtr;

begin
  New(P);
  P^.Next := nil;
  P^.Count := 23;
  P^.Name := #2'Jones';
end.
```

und lassen es (via F4) bis zum abschließenden **end** des Hauptprogramms ausführen. Danach läßt sich die Wirkung expliziter Formatangaben bei *Debug/Evaluate* recht anschaulich demonstrieren:

Angabe	dargestellter Wert
List	(10, 20, 30, 40, 50)
List[2],3	20, 30, 40
List[2],3X	$14, $1E, $28
List, M	0A 00 14 00 1E 00 28 00 32 00
P	PTR($3AE0,$0C)

Angabe	dargestellter Wert
P,P	3AE0:000C
P^	(NIL;23;#2'Jones')
P^,RX	(NEXT:NIL;COUNT:$17;NAME:#$2'Jones')
P^.Name,C	'Jones'
P^,7M	(00 00 00 00 17 00 06)

Vorgaben durch die Position des Cursors im Quelltext

Wie in Teil I demonstriert, gibt Turbo Pascal im Feld *Evaluate* bzw. im Eingabefeld eines *Watch*-Ausdrucks das Wort an der Position des Cursors vor, wenn der Cursor auf einem Zeichen dieses Wortes in einem Quelltext steht; jede Eingabe von ⊡ kopiert ein weiteres Zeichen aus dem Quelltext in das Eingabefeld. Wenn sich beispielsweise im Quelltext eine Zeile wie

```
if Buffer[x+$80] = 'N' then ...
```

findet, dann läßt sich der Ausdruck *Buffer*[*x+$80*] recht einfach in das Eingabefeld von *Evaluate* oder *Watch* kopieren: Setzen Sie den Cursor auf eine beliebige Position innerhalb des Wortes *Buffer*, aktivieren Sie die Funktion (*Debug/Evaluate* bzw. *Debug/Watches/Add watch*) und drücken siebenmal auf ⊡.

Vorgegebene Ausdrücke werden im Eingabefeld hervorgehoben darstellt und wie alle anderen Vorgaben von Turbo Pascal bearbeitet: Wenn Sie als *erstes* einen Befehl zur Steuerung des Cursors benutzen (wie z.B. Home, End oder auch Backspace), dann bleibt die Vorgabe erhalten und kann weiter bearbeitet werden; geben Sie dagegen zuerst ein »normales« Zeichen ein, dann nimmt Turbo Pascal an, daß Sie eine komplett neue Eingabe starten wollen, und löscht die Vorgabe vollständig.

Wörter, die Turbo Pascal als Vorgabe liefern soll, müssen mit einem Buchstaben beginnen. Wenn der Cursor beispielsweise am Anfang eines Ausdrucks wie

```
23 + MyRec.XField * NPtr^.Nums
```

steht, wird dieser Text also nicht als Wort bewertet. In derartigen Fällen läßt sich der Ausdruck dennoch aus dem Quelltext kopieren: Setzen Sie einfach den Cursor auf die 2, aktivieren Sie die Funktion und verwenden Sie dann ⊡, um den Text ab der Cursorposition zeichenweise in das Eingabefeld zu kopieren. (⊡ funktioniert auch dann, wenn Turbo Pascal den Text ab der Cursorposition nicht als Wort erkennt).

Verändern von Werten über Debug/Evaluate

Über eine Eingabe im Feld *New value* läßt sich einem im Feld *Evaluate* eingegebenen Ausdruck ein neuer Wert zuweisen. Voraussetzung dafür ist, daß

- der in *Evaluate* formulierte Ausdruck auch in einem Pascal-Programm auf der linken Seite einer Zuweisung zulässig ist,
- und die Typen beider Ausdrücke miteinander zuweisungskompatibel sind.

Anders gesagt: dem Ausdruck

```
IntArray[x+80]
```

läßt sich ein neuer Wert vom Typ *Integer* zuweisen, dem Ausdruck

```
IntArray[x]+80
```

dagegen nicht, weil er nicht für eine einzelne Variable, sondern für das Ergebnis einer Rechenoperation steht. (Der Compiler würde den Versuch einer derartigen Zuweisung im Quelltext eines Programms mit der Meldung »Cannot be modified« quittieren – und genau das tut *Debug/Evaluate* in diesem Fall auch).

Mögliche Anwendungen dieser Funktion sind unter anderem das »Überspringen« von Fehlern (d.h. die manuelle Korrektur falscher Funktionsergebnisse oder Variablenwerte) und die Prüfung von Programmteilen auf Grenzbedingungen hin – über die direkte Veränderung von Parametern lassen sich recht einfach Programmzustände simulieren, deren Simulation in den meisten Fällen wesentlich umfangreichere Manipulationen notwendig machen würde.

»Qualifizierte« Bezeichner

Eine entsprechende Angabe vorausgesetzt, kann der Debugger auch auf lokale Variablen und typisierte Konstanten zugreifen, die außerhalb des Geltungsbereichs der momentanen Routine liegen. Die dazugehörige Syntax arbeitet nach denselben Prinzipien wie bei Units (vgl. Kapitel 3 »Units, Overlays und andere Geheimnisse« dieses Bandes und Kapitel 4 »Variablen« des *Programmierhandbuches*):

```
[Modulname.][Routinenname.][Routinenname. ...][.Variablenname]
```

Je nachdem, auf welcher »Ebene« sich der Bezeichner befindet, muß nur ein Teil dieser Elemente angegeben werden:

`XProc.LVar` bezeichnet eine lokale Variable der Routine *XProc*

`XProc.InXProc.LVar` bezeichnet eine lokale Variable der Routine *InXProc*, die ihrerseits lokal zur Routine *XProc* deklariert ist

`XUnit.XProc.LVar` bezeichnet eine lokale Variable einer Routine *XProc*, die in der Unit *XUnit* definiert ist

Die Angabe des Unitnamens bei Bezeichnern, die innerhalb des jeweiligen Moduls global gültig sind, ist nur notwendig, wenn dieser Name von mehreren Units für unterschiedliche Variablen verwendet wird.

Teil III: Grenzbedingungen und Programmentwicklung

Nachdem die Befehle und Möglichkeiten nun wirklich erschöpfend behandelt sein sollten, geht es in diesem Teil um die praktische Anwendung: zum einen um die technischen Grenzen des Debuggers; zum anderen darum, wie

man sich das Leben bereits bei der Entwicklung des Programms ein wenig leichter machen kann.

Programmierung »mit dem Debugger«

Wenn einige grundlegende Regeln bei der Programmierung eingehalten werden, fällt eine spätere Fehlersuche auch vom technischen Standpunkt wesentlich einfacher aus. Zur Erklärung des Wie und Wo müssen wir allerdings erst einmal kurz auf die Hintergründe des Debuggers eingehen:

Wenn ein Programm mit *O/C/Debug information..On* compiliert wird, dann speichert Turbo Pascal zwei Tabellen, die letztendlich dieselben Informationen wie eine .MAP-Datei enthalten: zum einen die Adressen und Namen aller globalen Bezeichner; zum anderen die »Zeilennummern« des Quelltextes zusammen mit den jeweiligen Adressen im Codesegment des Programms.

Wenn der Schalter *O/C/Local symbols* ebenfalls auf *On* gesetzt ist, dann kommt noch eine dritte Tabelle hinzu, die Namen lokaler Variablen und ihre relative Lage auf dem Stack bei Aktivierung der jeweiligen Routine aufnimmt.

Vor diesem Hintergrund betrachtet, dürften einige Dinge fast selbsterklärend sein:

- Lokale Variablen existieren nicht nur vom logischen Standpunkt ausschließlich dann, wenn die entsprechende Routine aktiv ist – ihre Adresse besteht sowohl für das Programm als auch den Debugger aus einem relativen Abstand zur »momentanen« Spitze des Stacks. Folglich sind ihre Werte außerhalb der jeweiligen Routine undefiniert – woran auch die Angabe eines qualifizierten Bezeichners nichts ändert.

- Lokale typisierte Konstanten haben dagegen einen festen Platz im Datensegment des Programms – ihre Werte lassen sich über die Angabe qualifizierter Bezeichner auch außerhalb der jeweiligen Routine ermitteln.

- Lokale Routinen haben einen festen Platz im Codesegment des Programms, dazu kommen die »Zeilennummern« des Quelltextes – deshalb lassen sie sich auch dann schrittweise verfolgen, wenn der Schalter *Local symbols* bei der Compilierung auf *Off* gesetzt war. Die Ausgabe der Adresse einer lokalen Routine mit *Evaluate* läuft dagegen über den *Bezeichner* – und der wird nur dann gespeichert, wenn *Local symbols* auf *On* gesetzt ist.

- Breakpoints und die schrittweise Ausführung mit *Step over* bzw. *Trace into* orientieren sich an den »Zeilennummern« des Quelltextes. Bei einer Programmzeile wie

```
x:= 10; y := 20; z := 30;
```

verfügt der Debugger nur über die »Startadresse der Zeile« im Codesegment – was bedeutet, daß diese drei Zuweisungen grundsätzlich in *einem* Schritt ausgeführt werden. Wenn wir dieselben Befehle in voneinander getrennten Zeilen formulieren, also

```
x := 10;
y := 20;
z := 30;
```

dann hat der Compiler drei »Zeilenadressen« gespeichert – folglich läßt sich jede Zuweisung als Einzelschritt ausführen, nach dem das Programm wieder angehalten wird. Anders gesagt: der Debugger folgt der »optischen Logik« eines Programms – je exakter das Listing den tatsächlichen Ablauf widerspiegelt, desto leichter wird man sich bei der Verfolgung einzelner Schritte tun.

Rekursive Routinen

Rekursion wird die Progammiertechnik genannt, bei der eine Routine sich - direkt oder indirekt - selbst aufruft. Im folgenden Beispiel ruft sich die Funktion *Power* selbst auf, um den benötigten Rückgabewert zu berechnen:

```
program TestPower;

function Power (Base, Exp: Word): Longint;
begin
   if Exp <= 0 then
      Power := 1
   else
      Power := Base * Power (Base, Exp-1);
end;   { von Power }

begin  { Beginn des Hauptprogramms }
   Writeln('2 hoch 14 = ',Power(2,14));
end.   { von TestPower }
```

Beim Debuggen von rekursiven Routinen müssen Sie auf einige Besonderheiten achten. Viele Rekursionsebenen verbrauchen eine Menge Systemstack, was manche unerwünschte Nebeneffekte zur Folge haben kann (z.B. Programmabbruch oder Systemabsturz aufgrund von Stacküberlauf). Es handelt sich hierbei um eine typische Gefahr beim Einsatz von rekursiven Programmiertechniken. Stürzt Ihnen ein solches Programm während des Debuggens ab, so dürfte dies in den meisten Fällen auf einen Stacküberlauf zurückzuführen sein.

Die sofortige Rückkehr aus einer sehr tiefen Rekursionsebene mit Hilfe des Return-Stacks zu einem gewünschten Punkt ist nicht immer möglich. Dies liegt an der Begrenzung des Return-Stacks auf die letzten 128 Funktions-/Prozeduraufrufe. Sie können allerdings vom unteren Ende des Return-Stacks so weit wie möglich zurückspringen und anschließend den Return-Stack erneut benutzen.

Bei jedem Aufruf einer rekursiven Funktion wird ein neuer Satz lokaler Variablen und call-by-value (nicht-**var**) Parameter angelegt. Sollten Sie den Wert einer derartigen Variablen in einem *Watch*-Fenster verfolgen, müssen Sie daran denken, daß der gezeigte Wert zur jeweils aktiven Aufrufebene gehört.

Die Grenzen des Arbeitsprinzips

Zur Speicherung von »Zeilenadressen« und Befehlen gibt es eine Ausnahme, die aber nur scheinbar eine ist: **inline**-Routinen und -Befehle werden entsprechend der Pascal-Syntax als einzelner Befehl behandelt – unabhängig davon, über wieviele Zeilen des Quelltextes sie sich erstrecken. Da der Compiler via **inline** eingefügte Bytefolgen in keiner Weise analysiert (und deshalb keine Informationen über »Zeilenadressen« liefern kann), werden sie auch vom Debugger als »unteilbar« betrachtet und immer in einem einzigen Schritt ausgeführt. Mit gutem Grund: die Syntax von **inline** läßt es ohne weiteres zu, *einen* Prozessorbefehl über mehrere Zeilen des Quelltextes zu verteilen – und jeder Versuch, den Prozessor *in* einem Befehl zu unterbrechen, dürfte entweder wirkungslos bleiben oder bestenfalls unerfreuliche Folgen nach sich ziehen. (Technisch: der Prozessor würde einen vom Debugger an der falschen Stelle eingefügten Interrupt-Befehl entweder als Datum interpretieren und somit faktisch ignorieren oder – wesentlich schlimmer – es käme irgendein undefinierter Befehlscode dabei heraus, der das System in die ewigen Jagdgründe schickt).

Ähnliches gilt für Routinen, die als **external** eingebunden oder mit *O/C/Debug information..Off* compiliert wurden: hier fehlen Debugger und Compiler ebenfalls die notwendigen Informationen – folglich werden sie *en bloc* ausgeführt.

Die Routine, die am Ende eines Programms den Aufruf zuvor gesetzter Exit-Prozeduren abwickelt, ist ein Bestandteil der Bibliotheken und als solche für den Debugger nicht zugänglich: Deshalb läßt sich der *Aufruf* von Exit-Prozeduren auch nicht mit dem Debugger verfolgen.

Hier gibt es allerdings einen recht einfachen Ausweg: Wenn Sie auf das **begin** der jeweiligen Routinen einen Breakpoint setzen, dann bekommt der Debugger bei der *Ausführung* wieder die Kontrolle – und weil es sich bei den Exit-Prozeduren selbst um Teile des Pascal-Programms handelt, ist ein schrittweises Fortsetzen dann problemlos möglich. Ein Beispiel dazu:

```
program ExitTrace;

var
  ExitSave: Pointer;  { für die originale Exit-Prozedur }

{$F+} procedure MyExit; {$F-}
begin
  Writeln('Hier ist MyExit');  { ein Lebenszeichen }
  ExitProc := ExitSave;   { ExitProc wieder aufs Original }
end;

begin
  ExitSave := ExitProc;  { Original-Zeiger speichern }
  ExitProc := @MyExit;   { und auf die eigene Routine setzen }

  Writeln('Hier ist das Hauptprogramm');
end.
```

Wenn Sie hier versuchen, das abschließende **end** des Hauptprogramms mit *Trace into* zu verfolgen, endet das Programm mit der Ausführung von *MyExit*,

ohne daß der Debugger noch einmal die Möglichkeit zum Eingreifen gehabt hätte. Anders sieht die Sache dagegen aus, wenn ein permanenter Breakpoint auf dem **begin** von *MyExit* gesetzt ist – in diesem Fall bekommt der Debugger die Kontrolle zurück und ermöglicht eine schrittweise Ausführung der Routine. (Eine weitere Möglichkeit wäre, den Cursor im Quelltext auf das **begin** von *MyExit* zu setzen und das Programm dann via *Go to cursor* bis zu diesem Punkt ausführen zu lassen – auch das funktioniert ohne jedes Problem).

Im folgenden sind alle Fälle aufgelistet, bei denen der Debugger nicht oder nur eingeschränkt benutzt werden kann:

- Bei **inline**-Funktionen oder Prozeduren;
- bei Turbo-Pascal-Routinen aus einer der Standard-Units (*Crt, Dos, Graph, Graph3, Overlay, Printer, System, Turbo3*);
- bei **external**-Funktionen oder Prozeduren;
- bei **interrupt**-Funktionen oder Prozeduren;
- bei Prozeduren, Funktionen oder Initialisierungscode, die sich in Units befinden, die nicht mit der {$D+}-Direktive (oder mit *Options/Compiler/Debug information* auf *On*) compiliert wurden;
- bei Prozeduren, Funktionen oder Initialisierungscode, die sich in Units befinden, deren Quelltext nicht verfügbar ist. Falls dieser sich nicht im Unit- oder im aktuellen Verzeichnis befindet oder wenn der zugehörige Dateiname anders als *unitname*.PAS lautet (wobei *unitname* der in **uses** verwendete Name ist), werden Sie von der IDE nach dem richtigen Namen gefragt. Falls Sie dabei einen Null-Dateinamen eingeben oder Esc drücken, fährt der Debugger so fort, als wäre keine Debugger-Information vorhanden;

...und die Grenzen des Hauptspeichers

Bei großen Programmen ist es sehr wohl möglich, daß Ihnen bei der Arbeit mit dem Debugger der Speicherplatz ausgeht: schließlich drängeln sich hier Betriebssystem, Editor, Compiler, Debugger, der momentane Quelltext, das ausführbare Programm zusammen mit seinen Daten und die Symboltabellen des Debuggers – und das alles zur selben Zeit. Wieviel Platz Ihnen noch bleibt, läßt sich jederzeit (auch während der Aktivitäten des Debuggers) über *File/Get info* feststellen – falls die Anzeige *Free Memory* gefährlich nahe an Null rutscht (oder Sie bereits die lakonische Meldung *Out of memory. Press ESC* erhalten haben), dann sollten Sie die folgende Checkliste durchgehen.

- Verkleinern Sie die von der Entwicklungsumgebung voreingestellten Speichergrößen für Fenster, Editor und des Overlay-Puffers. Diese Einstellungen können entweder im Menü *Options/Environment/Startup* oder über die entsprechenden Kommandozeilen-Parameter gemacht werden. Dabei gibt es folgende Puffer, die Turbo Pascal's integrierte Entwicklungsumgebung benutzt:
  - *Window heap size* legt die Puffergröße für geöffnete Fenster fest. Wertebereich: 32 bis 64 KB; Standard-Einstellung: 32 KB; Kommandozeilen-Parameter: /W
  - *Editor heap size* gibt die Puffergröße für Texte in Editorfenstern an. Werte-

bereich: 28 bis 128 KByte; Standard-Einstellung: 28 KByte; Kommandozeilen-Parameter: /E

- *Overlay heap size* ist die Größe des Overlay-Puffers, den die Entwicklungsumgebung für ihre Overlays verwendet. Wertebereich: 64 bis 256 KByte; Standard-Einstellung: 112 KByte; Kommandozeilen-Parameter: /O

Bei den Kommandozeilen-Parametern ist die Angabe der Speichergröße in KByte. D.h. /E64 veranlaßt die Entwicklungsumgebung beim Start, 64 KByte als Editor-Puffer zu reservieren.

Verfügt Ihr Rechner über EMS-Speicher, läßt sich der Puffer für Overlays auf den Minimalwert einstellen ohne daß sich die Arbeitsgeschwindigkeit der integrierten Entwicklungsumgebung verringert. Dies gilt allerdings nur, wenn im Menü *Options/Environment/Startup* die Verwendung von Expanded memory (EMS) eingestellt worden ist. Mit dem Kommandozeilen-Parameter /X erreicht man das gleiche Ergebnis.

- Entfernen Sie unbenötigte Units aus der Datei TURBO.TPL. Diese Bibliothek wird beim Start der integrierten Entwicklungumgebung in den Speicher geladen, um die Übersetzungsgeschwindigkeit zu erhöhen. Man kann Turbo Pascal auch anweisen, diese Bibliothek überhaupt nicht in den Speicher zu laden. Da diese Datei über 40 KByte groß ist, kann dies eine beträchtliche Vergrößerung des freien Speichers bringen. Dabei ist allerdings zu beachten, daß die Datei SYSTEM.TPU, die von jedem Programm benötigt wird, aus der Datei TURBO.TPL mittels TPUMOVER extrahiert wurde. Der entsprechende Kommandozeilen-Parameter ist /T.

  Achtung: Die Einstellungen im Menü *Options/Environment/Startup* werden erst nach einem Neustart aktiv. Um also über dieses Menü etwas zu verändern, müssen Sie die Entwicklungsumgebung erst verlassen und wieder starten.

- Compilieren Sie Ihr Programm mit *Compile/Destination...Disk*. (Dieser einfache Tricks spart mitunter einige Dutzend Kilobyte im RAM – nicht zuletzt deshalb, weil der Linker beim Erzeugen einer .EXE-Datei unbenötigte Routinen der Standard-Units zusammen mit überflüssigen Datendeklarationen entfernt).

- Arbeitet Ihr Programm ohne dynamische Variablen: Setzen Sie die Minimal- und Maximalgröße des Heaps (via *Options/Compiler/Memory sizes* oder durch den Compiler-Befehl {$M xx,yy,zz} im Quelltext) auf 0. Analog dazu können Sie es auch mit einer Verkleinerung des Stacks probieren – die Standardvorgabe von 16 KByte ist für die meisten Programme recht großzügig bemessen.

- Entfernen Sie speicherresidente Programme (wie SideKick Plus, TPCED, SuperKey usw).

- Compilieren Sie eine oder mehrere Units Ihres Programms im Modus {$L-} (d.h. ohne Informationen über lokale Symbole). Wie im vorangegangenen Abschnitt erläutert, lassen sich die Routinen dieser Units auch dann noch schrittweise verfolgen; Sie nehmen dem Debugger lediglich die Möglichkeit, lokale Variablenwerte auszugeben und/oder zu verändern.

- Compilieren Sie bereits getestete Units im Modus {$D-} (d.h. ohne jede Information zur Fehlersuche). Die entsprechenden Programmteile verhalten

sich danach für den Debugger wie Routinen der Laufzeitbibliothek – sie werden grundsätzlich *en bloc* ausgeführt.

- Compilieren Sie das gesamte Programm (oder Teile davon) mit {$R-,S-}, d.h. ohne die automatische Prüfung von Bereichen und des Stacks. Der Compiler erzeugt für jede Prüfung einige zusätzliche Bytes Code, die sich damit einsparen lassen (was entsprechende Sorgfalt bei der späteren Prüfung des Programms voraussetzt, damit diese Einsparung nicht zum Bumerang wird).

- Versuchen Sie, einige Units Ihres Programms (mit der Aufnahme von *Overlay* und entsprechenden {$O}-Befehlen) als Overlay zu formulieren. Diese Maßnahme dürfte mit am aufwendigsten sein, verspricht aber auch den größten Erfolg: eine entsprechende Organisation vorausgesetzt, fallen damit sämtliche Schranken für die Programmgröße. Details zu Overlays finden Sie in Kapitel 3 »Units, Overlays und andere Geheimnisse« sowie in Kapitel 12 »Standard-Units« des *Programmierhandbuches*.

- Versuchen Sie, einzelne Units separat zu bearbeiten, d.h. reduzieren Sie Ihr Programm auf das Grundgerüst, das zur Prüfung einer bestimmten Unit unbedingt notwendig ist. (In manchen Fällen dürfte dieser Vorschlag schwierig bis unmöglich in die Praxis umzusetzen zu sein – fast immer erzeugt aber allein der Versuch bereits ein übersichtlicheres und besser strukturiertes Programm).

Damit sind wir am Ende unserer Weisheit angelangt: Wenn Sie die Vorschläge dieser Checkliste in die Tat umgesetzt haben und der Platz im Hauptspeicher immer noch nicht reicht (was bei einer Unterteilung in Overlays allerdings eher unwahrscheinlich ist), dann bleibt Ihnen (neben einem eigenständigen Programm wie dem *Turbo Debugger*) nur noch ein Rückgriff auf die schlechten alten Zeiten: compilieren Sie Ihr Programm im Modus {$D-} und versuchen Sie, eventuelle Fehler über Ausgaben mit *Writeln* einzukreisen (wozu man allerdings wieder Befehle zur bedingten Compilierung verwenden könnte – siehe Kapitel 2 »Compiler-Befehle« des *Referenzhandbuches*).

Zum wirklich effektiven Prüfen von Programmen...

ist ein guter Debugger (und eine gewisse Übung damit) natürlich eine der wichtigsten Grundlagen – aber leider bei weitem noch keine Garantie für den Erfolg. Nachdem Alan Turing – einer der Pioniere der Informatik – bereits vor rund 30 Jahren schlagend bewiesen hat, daß es *nie* möglich sein wird, ein Programm durch ein anderes Programm vollständig analysieren zu lassen, gehört die Forschung nach den Ursachen, wann, wo, wie und warum sich ein Programm unerwartet verhalten bis unfreundlich benehmen könnte, immer noch zu den ungelösten Herausforderungen der Computerwissenschaft.

Allerdings kann man sich die Sache zumindest etwas leichter machen, wenn man einige allgemeine Regeln beherzigt – und um diese Regeln geht es in den folgenden Abschnitten.

Ein standardisierter Ansatz

wäre das erste, was man sich zurechtlegen sollte: Eine Checkliste mit einzelnen Schritten, die aller Erfahrung nach zu einem zuverlässigen Programm führen.

Natürlich gibt es hier keinen »richtigen« Weg (sonst hätten wir Alan Turing vielleicht doch noch widerlegt) – wie diese Checkliste aussieht, hängt vom Typ der Programme ab, die Sie schreiben; von Ihren Stärken und Schwächen als Programmierer – und natürlich vom persönlichen Stil, an eine Aufgabe heranzugehen. Die folgende Aufstellung sollte daher eher als eine Art Grundstein für eigene Wege gesehen werden.

- Nachdem die offensichtlichen Fehler beseitigt sind: lassen Sie das Programm Daten verarbeiten, die zwar nicht trivial, aber auf jeden Fall noch einfach und übersichtlich sind. Verfolgen Sie einzelne Programmteile, die Ihnen kritisch erscheinen, und verwenden Sie *Debug/Evaluate* zur Prüfung auch von Variablenwerten, die »eigentlich sowieso stimmen«.

- Versuchen Sie als nächstes, Datensätze zu konstruieren, mit denen noch nicht geprüfte Programmteile erreicht und ausgetestet werden können.

- Untersuchen Sie jeden einzelnen Befehl. Diese Empfehlung ist ernst gemeint: Vielleicht finden Sie dabei Fehler, an deren Existenz Sie nicht einmal gedacht haben. (Was hat eigentlich die **if**-Bedingung in der **while**-Schleife unseres Beispielprogramms *IndexFault* zu suchen?)

- Achten Sie besonders auf Befehle, die Verzweigungen enthalten und deshalb mit mehreren Datensätzen geprüft werden müssen. In einem an sich einfachen Vergleich wie

```
if XStr > YStr then ...
```

gibt es nicht nur *drei* mögliche Ergebnisse (*XStr=YStr, XStr<YStr, XStr>YStr*) und damit mindestens drei verschiedene Datensätze, die notwendig sind, um jedes dieser Ergebnisse zuverlässig zu prüfen – durch die etwas merkwürdige Codierung der deutschen Umlaute als »Grafikzeichen« wird beispielsweise der String *'Ärgerlich' hinter 'ZZZZZZ'* eingeordnet.

- Gehen Sie sämtlichen »Grenzbedingungen« nach: Wann verläßt das Programm eine Schleife, bis zu welchem Index werden die Elemente eines Arrays gesetzt? Fehler scheinen eine besondere Vorliebe für die Grenzen von Programmen zu haben, die manchmal eben doch schneller und häufiger erreicht werden, als es sich der Programmierer gedacht hatte.

- Schalten Sie den Debugger einstweilen ab, und prüfen Sie (mit Eingaben, Ausgaben, Optionen usw.), ob sich das Programm so benimmt, wie es der Benutzer erwartet. Wenn das Programm von anderen benutzt werden soll: prüfen Sie jede Reaktion (auch auf unsinnige Eingaben, Ausgaben, Optionen usw). Reagiert es in jedem denkbaren Fall vernünftig (oder sind die Reaktionen zumindest exakt reproduzierbar?)

■ Wenn Sie das Programm nicht ausschließlich zur persönlichen Verwendung geschrieben haben, sollte es mindestens von einer anderen Person getestet werden, die über zwei spezielle Eigenschaften verfügen sollte: zum einen muß sie in ihren Bedürfnissen (und ihrem Wissensstand in Sachen »Computer«) den potentiellen Benutzern entsprechen; zum anderen wird von ihr genug Zeit und Beharrungsvermögen verlangt, auch exotischere Dinge auszuprobieren bzw. eventuelle Probleme akkurat festzuhalten. Lassen Sie das Programm nur dann von einem anderen Programmierer testen, wenn es auch wirklich für Programmierer gedacht ist.

Modifikationen verlangen doppelte Aufmerksamkeit

Wenn Sie ein Programm nachträglich verändern (oder auch nur korrigieren), darf sich die Prüfung nicht unmittelbar auf die neugeschriebenen Teile beschränken. Welche Module hängen von den veränderten Programmteilen ab?

Je komplexer und größer ein Programm ist, desto mehr Arbeit hat man mit der Dokumentation von Änderungen – und desto wichtiger ist sie. Halten Sie zusätzlich alle durchgeführten Tests (und eventuell dazu verwendeten Datensätze) fest, damit diese Prüfungen bei weiteren Modifikationen ohne größeren Aufwand erneut durchgeführt werden können.

Defensiver Programmierstil

Nehmen Sie sich die Zeit für »saubere« Listings mit deskriptiven Variablennamen, Einrückung der Ebenen und ausreichender Kommentierung.

Halten Sie den Programmcode so einfach wie möglich. Mit den Möglichkeiten zur expliziten Typ-Umwandlung und anderen Freiheiten, die Turbo Pascal bietet, ist es keine allzugroße Schwierigkeit, ein halbes Dutzend an sich einfacher Operationen zu einem wahrhaften Monstrum mit einer einzigen Zielvariablen werden zu lassen – tatsächlich optimiert der Compiler den Code aber so weit, daß man sich derartige Operationen in fast allen Fällen sparen kann.

Versuchen Sie, Ihr Programm aus einfachen und gut dokumentierten Routinen heraus aufzubauen. Dadurch werden nicht nur Prüfungen vereinfacht, sondern auch eventuell notwendige Modifikationen zu einem späteren Zeitpunkt.

Jede Routine sollte soweit wie möglich in sich geschlossen sein und mit einem absoluten Minimum an globalen Daten auskommen. Durch diesen Ansatz (die sogenannte lockere Kopplung) werden nicht nur Prüfungen und spätere Modifikationen vereinfacht – auch das Durcheinander, das eine in sich geschlossene Funktion im Ernstfall anrichten kann, wird dadurch in Grenzen gehalten.

Optimieren Sie nicht während des Programmentwurfs. Schreiben Sie Ihr Programm als erstes so übersichtlich wie möglich, prüfen Sie sein Verhalten... Wenn sich dann herausstellen sollte, daß einige Funktionen zeitlich optimiert werden

müssen, haben Sie im Idealfall ein gut funktionierendes Programm zum »Tuning« vor sich – im Gegensatz zu einer Sammlung kryptischer Routinen, die zwar die zeitlichen Anforderungen erfüllen, aber noch Fehler enthalten und sich wegen der »Optimierung« an der Grenze der Lesbarkeit bewegen.

Fassen Sie generalisierbare Abläufe in Routinen zusammen, die nicht nur mehrfach innerhalb dieses Programms, sondern eventuell auch in anderen Projekten verwendet werden können. Die Erstellung und die Fehlersuche in einem generalisierten Modul ist in den meisten Fällen weniger zeitaufwendig als derselbe Arbeitsablauf mit zwei oder drei Modulen, die sich nur in Kleinigkeiten unterscheiden.

Fehlersuche von »unten nach oben«

Soweit irgendwie möglich, sollten Sie sich bei der Suche nach Fehlern zuerst auf die »unterste Ebene« konzentrieren (d.h auf Routinen, die ihrerseits keine Aufrufe weiterer Funktionen bzw. Prozeduren des Programms enthalten) und sich von dort aus in Richtung auf das Hauptprogramm vorarbeiten. Nur so können Sie sicher sein, daß sich Fehler nicht gerade in den Programmteilen verstecken, die Sie gerade mit *Step over* in einem Schritt ausgeführt haben.

Suche nach gleichen und ähnlichen Fehlern

Wenn Sie einen bestimmten Fehler lokalisiert haben, sollten Sie den restlichen Quelltext des Programms nach Wiederholungen oder Ähnlichkeiten absuchen. Ein Aufruf wie

```
Delete(XStr, Length(XStr)-1, 1); {l´scht das letzte Zeichen von XStr}
```

der korrekt eigentlich

```
Delete(XStr, Length(XStr), 1);
```

heißen muß – sollte Grund genug sein, über die Suchfunktion des Editors auch alle anderen Verwendungen von *Delete* (und eventuell *Insert*) kurz zu überprüfen.

7. Die IDE-Referenz

Programmieren mit Turbo Pascal ist ebenso einfach wie effizient. Alle Befehle zum Schreiben, Editieren, Compilieren, Linken und Debuggen können Sie mit nur einem Tastendruck ereichen. Dafür sorgt die *integrierte Entwicklungsumgebung* (Integrated Developement Enviroment), kurz IDE genannt.

Die neue Entwicklungsumgebung von Turbo Pascal ist mit zusätzlichen Extras ausgestattet, damit Sie sich bei der Arbeit mit Turbo Pascal auf Ihre wesentliche Aufgabe, das Schreiben fehlerfreier Programme, konzentrieren können:

- Mehrere verschiebbare Fenster, die beliebig verkleinert bzw. vergrößert werden können
- Mausunterstützung
- Dialogfenster
- Ausschneide- und Einfüge-Kommandos zum Kopieren von Textfragmenten (auch zwischen einzelnen Editorfenstern und aus Hilfe-Fenstern)
- Kommandos zum Suchen und Ersetzen
- Möglichkeit zum Ausdrucken
- Makro-Sprache für den Editor

Dieses Kapitel faßt für Sie in überschaubarer Weise zusammen, was Sie für den Start und das Verlassen von Turbo Pascal wissen müssen, es enthält außerdem eine detaillierte Zusammenstellung über die einzelnen Menü-Befehle, Dialogfenster, Schalter usw. Falls Sie sich jedoch zunächst mit den grundlegenden Komponenten der IDE beschäftigen möchten, dann lesen Sie Kapitel 1 »Die neue integrierte Entwicklungsumgebung« das Ihnen einige grundlegende Informationen über die IDE gibt und Sie dann in die Programmierung in der IDE einführt. Oder starten Sie TPTOUR. Dieses interaktive Tutorial emuliert die IDE von Turbo Pascal und zeigt Ihnen, wie man Programm-Dateien öffnet, diese editiert, compiliert, startet und wie man Fehler sucht. Außerdem werden Sie mit den allgemeinen Fähigkeiten der Fensterverwaltung vertraut gemacht. Sie können auch das Online-Hilfesystem von Turbo Pascal benutzen. In der Entwicklungsumgebung erhalten Sie mit einem Druck auf die F1-Taste Informationen über die IDE. Spezifische Hilfe zu allen Fragen, die Sprachimplementation von

Turbo Pascal betreffend, erhalten Sie mit Ctrl F1 innerhalb eines Editorfensters.

Aufrufen und Verlassen von Turbo Pascal

Wechseln Sie in das Verzeichnis mit Turbo Pascal, und rufen Sie in der DOS-Kommandozeile Turbo Pascal mit dem Befehl *TURBO* auf. Turbo Pascal gestattet Ihnen beim Aufruf die Angabe verschiedener Optionen als Parameter z.B. für die Verwendung zweier Monitore, eines LCD-Bildschirms, die Nutzung von Expansionsspeicher, von RAM-Disks und der EGA-Palette und vieles mehr.

Kommandozeilen-Optionen

Die Kommandozeilen-Optionen für die IDE von Turbo Pascal sind: /c, /d, /e, /g, /l, /n, /o, /p, /sx, /t, /w und /x. Die Angabe dieser Optionen muß folgender Syntax entsprechen:

```
turbo [Optionen] Dateinamen
```

Ein Plus- (+) oder auch ein Leerzeichen nach einer Option schaltet sie ein, ein Minuszeichen (-) schaltet sie ab. Zum Beispiel:

```
turbo /d /x- testprog
```

Die Angabe von /d veranlaßt Turbo Pascal, im Dual-Monitor-Modus zu arbeiten, /x- bewirkt, daß die IDE keinen Gebrauch vom Expansionsspeicher macht.

Die Option /c

Die Option /c gefolgt vom Dateinamen einer Konfigurationsdatei veranlaßt Turbo Pascal, die in dieser Datei gespeicherte Konfigurationseinstellung zu laden.

Die Option /d

Die Option /d veranlaßt Turbo Pascal, im Dual-Monitor-Modus zu arbeiten, falls die entsprechende Hardware zur Verfügung steht (z.B. eine Monochrom- und eine Farb-Graphikkarte); anderenfalls wird die Option /d ignoriert. Verwenden Sie den Dual-Modus, wenn Sie z.B. Ihr Programm innerhalb der IDE laufen lassen, es mit dem Debugger testen oder zur DOS-Kommandozeile via *File/DOS Shell* wechseln.

Verfügt Ihr Computer über zwei Monitore, behandelt DOS einen der beiden als den aktiven Monitor. Mit dem DOS-Kommando MODE können Sie zwischen den beiden Monitoren umschalten (z.B. MODE C080 oder MODE MONO). Im Dual-Monitor-Modus erscheint das Output-Fenster von Turbo Pascal auf dem

aktiven Monitor, der normale Turbo-Pascal-Bildschirm wird auf dem zweiten, nicht aktiven Monitor ausgegeben.

Das Umschalten zwischen aktivem und nicht aktivem Monitor erfolgt, wie gesagt, über das DOS-Kommando MODE, d.h. zum Wechseln des aktiven Monitors müssen Sie Turbo Pascal verlassen, den Wechsel vornehmen und Turbo Pascal mit der /d-Kommandozeilen-Option anschließend neu starten. Bei der Verwendung der /d-Option bedenken Sie bitte folgendes:

- Wechseln Sie den aktiven Monitor (z.B. mit dem DOS-Kommando MODE) nicht, wenn Sie temporär zur DOS-Ebene (*File/DOS Shell*) wechseln.
- Anwenderprogramme, die direkt auf die nicht aktive Bildschirmkarte zugreifen, werden nicht unterstützt und können unerwartete Ergebnisse produzieren.
- Verwenden Sie die Option Dual-Monitor (/d) nicht, wenn Sie Programme, die explizit mit zwei Monitoren arbeiten, mit dem Debugger testen oder laufen lassen.

Die Option /e

Die Option /e dient zum Einstellen des Editor-Heap. Die Standardeinstellung der Heap-Größe beträt 28 KByte und stellt auch gleichzeitig das Minimum dar; die maximale Größe des Editor-Heap beträgt 128 KByte. Die Vergrößerung des Editor-Heap dient lediglich dazu, die Arbeitsgeschwindigkeit der IDE zu erhöhen, wenn Sie mit langsamen Speichermedien für Swap-Dateien arbeiten. Lassen Sie die Standardeinstellung unverändert, wenn Ihnen Expansionsspeicher (EMS) zur Verfügung steht oder wenn Sie eine RAM-Disk (/s-Option) zum Speichern der Swap-Datei verwenden.

Die Option /g

Verwenden Sie die Option /g zum Speichern des Grafik-Bildschirms, wenn Sie auf einem EGA-, VGA- oder monochromen System ein Grafik-Programm mit dem Debugger untersuchen. Ist diese Option aktiviert, so reserviert die IDE zusätzliche 8 KByte des Hauptspeichers oder, soweit vorhanden, des EMS-Speichers für diesen Puffer.

Die Option /l

Die Option /l wird bei LCD-Bildschirmen verwendet.

Die Option /n

Die Option /n aktiviert bzw. deaktiviert eine spezielle Synchronisationsroutine für Bildschirmausgaben, die dazu dient, »Schneegestöber« auf dem Bildschirm zu unterbinden. Diese Option kommt nur dann zum Einsatz, wenn Ihr System mit einer CGA-Grafikkarte ausgerüstet ist. Falls Sie eine moderne CGA-Karte verwenden, auf der kein »Schnee« bei Bildschirmausgaben auftritt, sollten Sie diese Routine abschalten.

Die Option /o

Mit der Option /o wird die Heap-Größe des Overlay-Puffers der IDE verändert. Die Standardgröße des Puffers beträgt 112 KByte, das Minimum für den Puffer

beträgt 64 KByte und das Maximum 256 KByte. Falls Ihr Rechner mit EMS-Speicher ausgerüstet ist, können Sie die Größe des Overlay-Puffers bedenkenlos verringern, ohne dabei eine Verlangsamung der Arbeitgeschwindigkeit zu riskieren. Dafür steht mehr freier Speicher für die Compilierung und die Arbeit mit dem Debugger zur Verfügung.

Die Option /p

Die Option /p steuert die Umschaltung der Farbpalette bei EGA-Grafik-Karten, wenn Ihr Programm die Register der EGA-Palette verändert. Die EGA-Palette wird immer dann restauriert, wenn der Bildschirm gewechselt wird.

Normalerweise benötigen Sie diese Option nur dann, wenn Ihr Programm die Register der EGA-Palette verändert oder dafür die Grafik-Schnittstelle von Borland (BGI) verwendet.

Die Option /s

Verfügt Ihr Rechner nicht über EMS-Speicher, dann geben Sie mit der Option /s ein schnelles Laufwerk und ein Verzeichnis zum Auslagern der Swap-Datei an (z.B. eine RAM-Disk: /sd:\ ). Falls /s nicht spezifiziert ist, dann verwendet Turbo Pascal das momentan gesetzte Verzeichnis zum Auslagern der Swap- Datei.

Die Option /t

Durch die Angabe der Option /t verhindern Sie, daß Turbo Pascal beim Start die Datei TURBO.TPL in den Speicher lädt. Mit dieser Vorgehensweise erhöhen Sie die Kapazität der IDE. Wenn Sie mit dieser Einstellung arbeiten, dann muß die Unit *System* (SYSTEM.TPU) für Turbo Pascal gesondert zur Verfügung stehen (vgl. den Abschnitt über das Hilfsprogramm TPUMOVER in Kapitel 10 »Die Zusatzprogramme«)

Die Option /w

Mit der Option /w wird die Heap-Größe des Fenster-Puffers der IDE verändert. Die Standardgröße des Puffers beträgt 32 KByte, das Minimum für den Puffer beträgt 24 KByte und das Maximum 64 KByte. Durch die Verringerung des Heap des Fenster-Puffers erhöhen Sie den freien Speicher, der Ihren Programmen zur Verfügung steht, allerdings verringert sich dadurch die Zahl der Fenster, die gleichzeitig in der IDE geöffnet sein können. In der Regel brauchen Sie die Standardeinstellung des Fenster-Puffers nicht zu verändern.

Die Option /x

Mit der Option /x schalten Sie den Zugriff auf Expansionsspeicher (EMS) durch die IDE ab. Gemäß Standardeinstellung verwendet Turbo Pascal, soweit verfügbar, EMS-Speicher, um darin eigene Programmteile, die als Overlays realisiert sind, zwischenzuspeichern und damit die Arbeitsgeschwindigkeit zu erhöhen.

Verlassen von Turbo Pascal

Turbo Pascal können Sie auf zwei Arten verlassen.

- Mit *File/Exit* (oder Alt X) beenden Sie Turbo Pascal. Wenn Sie Änderungen nicht gespeichert haben, erscheint vor dem Verlassen von Turbo Pascal eine entsprechende Abfrage.

- Mit *File/DOS Shell* verlassen Sie Turbo Pascal zur DOS-Ebene. Turbo Pascal bleibt dabei im Speicher, die Kontrolle über den Rechner liegt jedoch wieder in der DOS-Ebene. Sie können jedes normale DOS-Kommando eingeben und andere Programme starten. Wenn Sie zu Turbo Pascal zurückkehren möchten, geben Sie in der DOS-Kommandozeile EXIT ein und drücken Enter . Daraufhin wird Turbo Pascal wieder aktiv und erscheint im selben Zustand wie zuvor.

System-Menü

Alt Leertaste

Das System-Menü erscheint als Symbol in der oberen linken Ecke des Menübalkens. Mit Alt Leertaste wählen Sie das Menü am schnellsten an. In dem Menü finden Sie mehrere allgemeine System-Kommandos (*About, Refresh Display* und *Clear desktop*).

About
öffnet ein Dialogfenster mit der Copyright- und der Versionsinformation von Turbo Pascal. Mit Esc , mit der Leertaste oder durch Anklicken von *OK* (oder Auswahl durch Enter schließen Sie das Fenster.

Refresh Display
restauriert den IDE-Bildschirm. Machen Sie von dieser Funktion Gebrauch, wenn Ihr Programm den IDE-Bildschirm überschrieben hat.

Clear Desktop
schließt alle Fenster und löscht alle Aufzeichnungslisten.

File-Menü

Alt F

Über das *File*-Menü können Sie von einem Editorfenster aus Dateien öffnen, neu anlegen, Veränderungen sichern, andere Dateifunktionen durchführen, in die DOS-Ebene wechseln und Turbo Pascal verlassen.

Open F3

Öffnet ein Dialogfenster zur Auswahl einer Datei, die Sie in ein Editorfenster laden können (siehe Abbildung 7.1).

Das Dialogfenster enthält ein Eingabefeld, eine Dateiliste und die Aktionsschalter *Open*, *Replace*, *Help* und *Cancel*. Außerdem erscheint immer ein Informationsbereich mit einer Kurzbeschreibung der angewählten Datei. Folgende Aktionen sind ausführbar:

- Geben Sie einen Dateinamen ein, und wählen Sie *Replace* oder *Open*. *Open* öffnet eine Datei und lädt diese in ein neues Editorfenster. *Replace* dagegen lädt eine Datei in ein bereits aktives Editorfenster und überschreibt damit den alten Inhalt dieses Fensters.

- Geben Sie einen Dateinamen mit Jokerzeichen (? und *) ein. Dadurch wird die Dateiliste aktualisiert und zeigt nur Dateien an, die Ihrer gewünschten Spezifikation entsprechen.

- Drücken Sie die →-Taste. Diese Aktion öffnet eine Eingabeaufzeichnungsliste, die von Ihnen bereits verwendete Dateispezifikationen anzeigt.

- Wählen Sie in der Dateiliste einen Verzeichnisnamen, um andere Verzeichnisse zu durchsuchen.

Ist keine Datei mit dem angegebenen Name vorhanden, dann legt Turbo Pascal automatisch eine solche Datei an und öffnet ein neues Editorfenster.

Das Informationsfeld im unteren Teil des Dialogfensters zeigt Pfad, Dateinamen, Datum, Zeit und Größe der ausgewählten Datei an. Die einzelnen Elemente sind hier reine Anzeigefelder und können nicht angewählt werden.

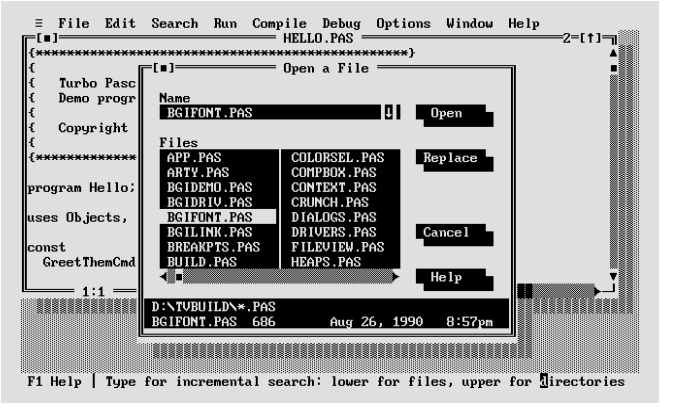


Abbildung 7.1 Das Dialogfenster File/Open

New

File/New öffnet ein Editorfenster und vergibt den Namen NONAMExx.PAS (xx steht für eine Zahl von 0-99). NONAME-Dateien sind temporäre Dateien. Beim Speichern dieser Dateien werden Sie aufgefordert, einen Dateinamen festzulegen.

Save F2

File/Save speichert die Datei im aktiven Fenster. Beim Speichern einer NONAME-Datei wird das Fenster *Save as* geöffnet, und Sie werden aufgefordert, einen Dateinamen festzulegen.

Save as

File/Save As ermöglicht Ihnen, die Datei im aktiven Editorfenster unter einem anderen Namen in ein anderes Verzeichnis und/oder auf ein anderes Laufwerk zu speichern.

Abbildung 7.2
Das Dialogfenster File/Save As

Save All

File/Save All arbeitet wie *Save*, speichert jedoch nicht nur die Datei im aktiven Editorfenster, sondern alle veränderten Dateien.

Change Dir

Mit *File/Change Dir* können Sie das aktuelle Laufwerk und das aktuelle Verzeichnis wechseln. Turbo Pascal verwendet das aktuelle Laufwerk/Verzeichnis zum Speichern von Dateien und für die Suche nach Dateien.

Abbildung 7.3
Das Dialogfenster Change Dir

Es gibt zwei Wege, Verzeichnisse zu wechseln: Geben Sie den Pfad ein, und drücken Sie Enter oder wählen Sie einen Pfad aus dem Verzeichnisbaum, und drücken Sie Enter oder klicken Sie *OK* an. Mit Esc brechen Sie die Aktion ab.

Nach der Auswahl von *OK* werden die Änderungen übernommen. Mit dem Schalter *ChDir* werden die Unterverzeichnisse des hervorgehobenen Verzeichnisses angezeigt. Bei aktiviertem *ChDir* hat Enter dieselbe Funktion wie doppeltes Anklicken. Sollten Sie Ihre Auswahl revidieren wollen, bevor Sie das Dialogfenster geschlossen haben, betätigen Sie den Schalter *Revert*.

Print

File/Print druckt den Inhalt des aktiven Editorfensters. Tabulatorzeichen werden dabei zu Leerzeichen erweitert. Mit Ctrl K P können Sie einen markierten Textblock ausdrucken.

Get Info

File/Get Info zeigt Informationen über die aktuelle Datei an.

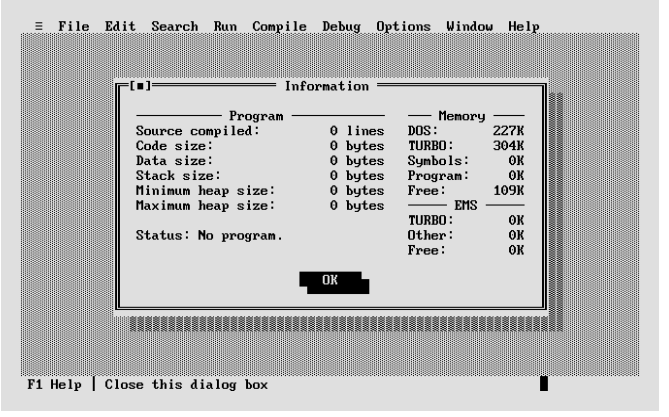


Abbildung 7.4 Das Dialogfenster Get Info

Das Fenster dient lediglich zur Anzeige und kann nicht verändert werden. Mit Enter können Sie das Fenster wieder schließen.

DOS Shell

File/DOS Shell verläßt Turbo Pascal vorübergehend und wechselt zur DOS-Ebene. Sie können DOS-Befehle eingeben oder andere Programme aufrufen. Durch die Eingabe von EXIT und Enter in der DOS-Ebene kehren Sie zu Turbo Pascal zurück. Zur Ausführung dieses Kommandos muß ausreichend Speicher zur Verfügung stehen. Sollten Sie gerade mit dem integrierten Debugger arbeiten, dann beenden Sie diese Arbeit mit *Run/Programm Reset* oder Ctrl F2, um zusätzlichen Speicher freizugeben.

Hinweise: Installieren Sie keine speicherresidenten Programme (wie z.B. Sidekick), wenn Sie mit *DOS Shell* auf die DOS-Kommandoebene gewechselt sind. Sie vermeiden dadurch Speicherprobleme.

Im Dual-Monitor-Modus erscheint die DOS-Kommandozeile auf dem Turbo-Pascal-Bildschirm. Sie können daher DOS anwählen, ohne die Programmaus-

gabe zu zerstören. (Da die Programm-Ausgabe auf einem der System-Bildschirme zur Verfügungs teht, sind *Window/User Screen* und Alt F5 deaktiviert.)

Exit Alt X

Mit dem Kommando *File/Exit* verlassen Sie Turbo Pascal und gelangen in die DOS-Ebene zurück. Turbo Pascal fordert Sie dann auf, noch nicht gesicherte Dateien abzuspeichern.

Edit-Menü

Alt E

Im Edit-Menü finden Sie Funktionen zum Ausschneiden, Kopieren und Einfügen von Text aus bzw. in ein Editorfenster. Außerdem können Sie das Fenster der Zwischenablage öffnen und deren Inhalt editieren.

Bevor Sie diese Kommandos anwenden, sollten Sie sich zunächst Kenntnisse über das Markieren von Text aneignen. Text kann sowohl über die Tastatur, als auch mit der Maus markiert werden – beide Arbeitsweisen führen zum selben Resultat, unterschiedlich ist lediglich ihre Ausführung:

- Markieren von Text über die Tastatur:
  - Drücken Sie Shift und eine der Cursortasten.
  - Mit Ctrl K B markieren Sie den Anfang eines Textblockes, mit Ctrl K K sein Ende.
  - Mit Ctrl K T markieren Sie das Wort, das durch die Cursorposition bestimmt ist.
  - Mit Ctrl K L markieren Sie eine Zeile.

- Markieren von Text mit der Maus:
  - Ziehen Sie den Mauszeiger über den Text (drücken Sie hierbei die linke Maustaste). Das Fenster bewegt sich über dem Text, wenn Sie eine Begrenzungslinie berühren.
  - Einzelne Zeile markieren: Klicken Sie die Zeile an einer beliebigen Stelle doppelt an.
  - Mehrere Zeilen markieren: Klicken Sie die erste Zeile zweimal an, und ziehen Sie die Maus über den Text.
  - Markierung erweitern/verkleinern: Drücken Sie Shift, und klicken Sie an einer beliebigen Stelle des Textes.

Ist der Text markiert, können Sie die Editor-Kommandos verwenden und mit der Zwischenablage arbeiten. Die Zwischenablage ist ein spezielles Fenster, das den Text verwaltet, der ausgeschnitten oder kopiert wurde, so daß Sie ihn wieder einfügen können.

Nun aber zu den Kommandos des Edit-Menüs:

Restore Line

Edit/Restore Line nimmt das letzte Editor-Kommando der zuletzt veränderten oder gelöschten Zeile zurück (betrifft z.B. auch Text, der in eine leere Zeile

eingegeben wurde, sowie das Kommando Ctrl Y . (*Restore Line* hat nur auf die zuletzt bearbeitete Zeile eine Wirkung.)

Cut Shift Del
Edit/Cut überträgt den markierten Text in die Zwischenablage und entfernt ihn dabei aus der Datei.
Mit *Paste* können Sie den Text an einer beliebigen Stelle in ein beliebiges Dokument einfügen. Der Text bleibt in der Zwischenablage, so daß Sie ihn mehrmals verwenden können.

Copy Ctrl Ins
Edit/Copy kopiert den markierten Text in die Zwischenablage, entfernt ihn im Gegensatz zu *Cut* dabei jedoch nicht aus der Datei (siehe auch *Cut* und *Paste*).

Paste Shift Ins
Edit/Paste fügt den in der Zwischenablage markierten Text in einem beliebigen Dokument an der Cursorposition ein (siehe auch *Cut* und *Copy*).

Copy Example
Edit/Copy Example kopiert ein Programmbeispiel aus dem aktuellen Hilfefenster in die Zwischenablage. Beispiele sind als kopierbare Blöcke definiert, d.h. sie müssen nicht extra markiert werden.

Show Clipboard
Edit/Show Clipboard öffnet das Zwischenablage-Fenster, das alle aus anderen Fenstern ausgeschnittenen und kopierten Textteile enthält. Hier können Sie das Textfragment markieren, das mit *Paste* eingefügt wird.
Das Zwischenablage-Fenster bietet Ihnen auch die Möglichkeit, die enthaltenen Textfragmente zu editieren.
Die Zwischenablage verhält sich wie ein Editorfenster, Kommandos wie Kopieren und Ausschneiden bewirken jedoch, daß der markierte Text immer an das Ende des Fensters plaziert wird.

Clear Ctrl Del
Edit/Clear löscht den markierten Text unwiderruflich, überträgt ihn also nicht in die Zwischenablage. Mit dieser Funktion ist es auch möglich, Textfragmente aus der Zwischenablage selbst zu löschen.

Search-Menü

Alt S

Über das Menü *Search* finden Sie Text, Prozedur- und Funktionsdeklarationen sowie fehlerhafte Stellen in Ihrem Programm.

Find Alt S F
Search/Find öffnet das Dialogfenster *Find* (siehe Abbildung 7.5). Hier können

Sie Suchtext eingeben und Optionen für die Suche setzen (alternatives Tastenkürzel: Ctrl Q F)

Das Dialogfenster *Find* enthält verschiedene Aktionsschalter, Schaltfelder und ein Markierungsfeld.

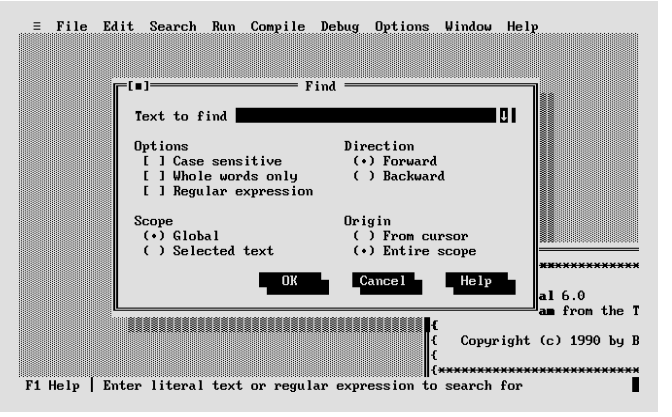


Abbildung 7.5 Das Dialogfenster Find

Das Markierungsfeld Options bietet folgende Einstellungen: Ist *Case sensitive* angegeben, wird wischen Groß- und Kleinschreibung unterschieden.

Mit *Whole words only* wird nach ganzen Wörtern gesucht (Wörter, die von Interpunktions- oder Leerzeichen umschlossen sind).

Mit *Regular expression* wird mit GREP-ähnlichen Jokerzeichen gesucht.

- `^` vor dem String: gibt den Zeilenanfang an.
- `$` am Ende des Ausdruckes: gibt das Zeilenende an.
- `.` steht für ein beliebiges Zeichen.
- `*` nach einem Zeichen: ersetzt beliebig viele Zeichen (inkl. kein Zeichen), z.B. *bo** findet *bot*, *b*, *boo*, *be*.
- `+` ersetzt beliebig viele Zeichen, eines ist aber mindestens notwendig; *bo+* findet *bot* und *boo*, aber nicht *be* und *b*.
- `[ ]` Zeichen in eckigen Klammern finden nur die angegebenen Zeichen, aber keine anderen; [*bot*] findet *b*, *o* oder *t*.
- `[^]` am Anfang des Strings: schließt die folgenden Zeichen von der Suche aus; [^*bot*] findet alle Zeichen außer *b*, *o*, *t*.
- `[-]` findet einen Zeichenbereich; [*b-o*] findet alle Buchstaben von *b* bis *o*.
- `\` vor einem Jokerzeichen: gibt an, daß das Zeichen buchstäblich genommen werden soll und nicht als Jokerzeichen interpretiert wird, z.B. steht `\^` für `^` und nicht für einen Zeilenanfang.

OK startet die Suche, *Cancel* verwirft die Eingaben wieder. ↵ zeigt die vorherigen Suchbegriffe an. Das Wort, auf dem der Cursor im Editorfenster steht, wird in der Eingabezeile als Vorgabe übernommen. Den Text, den Sie übernehmen möchten, erweitern Sie mit ↓.

Die Schaltfelder Direction, Scope und Origin Mit *Direction* geben Sie an, in welche Richtung gesucht werden soll. Stellen Sie vorher mit dem *Origin*-Schalter den Suchbeginn ein. Mit *Scope* geben Sie an, ob der ganze Text oder nur markierte Bereiche abgesucht werden sollen.

Mit *Origin* wird ab Cursorposition oder in der ganzen Datei in der mit *Direction* eingestellten Richtung und dem mit *Scope* ausgewählten Bereich gesucht.

Replace Alt S R

Search/Replace öffnet das Dialogfenster für das Suchen und Ersetzen von Text. Vergleichen Sie hierzu auch die Beschreibung zum Dialogfenster *Find* (Ctrl Q A ist ein alternatives Tastenkürzel für diese Funktion).

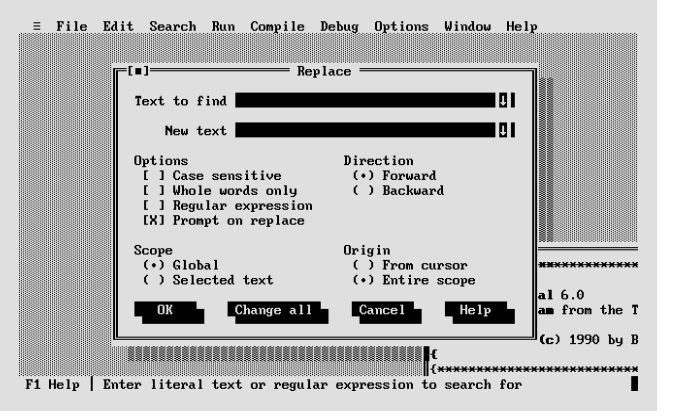


Abbildung 7.6 Das Dialogfenster Replace

Zusätzlich zum Dialogfenster *Find* finden Sie hier folgende Erweiterungen:

In das Eingabefeld *New Text* geben Sie den neu einzufügenden Text ein. *Prompt on Replace*, eine Erweiterung des Markierungsfeldes *Options*, bestimmt, ob vor jedem Ersetzen eine Rückfrage erscheinen soll. Der Aktionsschalter *OK* bewirkt, daß nur der erste gefundene Begriff ersetzt wird, durch den Aktionsschalter *Change All* werden alle Entsprechungen gemäß der Voreinstellung ersetzt. Ebenso wie im Fenster *Find* können Sie auch hier Text aus dem Editorfenster übernehmen.

Search Again Ctrl L

Search/Search Again wiederholt das letzte Suchen- oder Ersetzen-Kommando. Die Voreinstellungen werden dabei übernommen.

Go to Line Numer

Search/Go to Line Number sucht die eingegebene Zeilennummer. Turbo Pascal zeigt die aktuelle Zeilen- und Spaltennummer in der unteren linken Ecke eines jeden Editorfensters an.

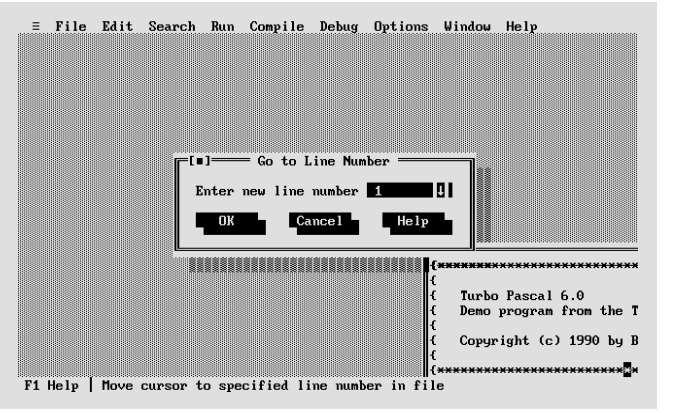


Abbildung 7.7 DasDialog fenster Go to Line Number

Find Procedure

Search/Find procedure öffnet ein Dialogfenster für die Suche nach einer Funktion oder Prozedur. Dieses Kommando arbeitet nur während eines Testlaufs mit dem Debugger.

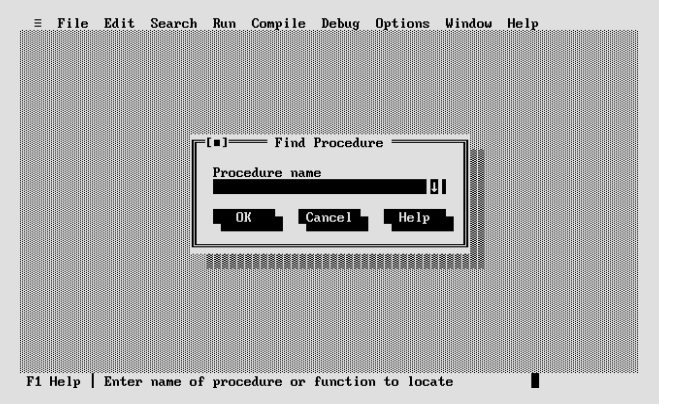


Abbildung 7.8 Das Dialog fenster Find Procedure

Geben Sie einen Prozedur- oder Funktionsbezeichner ein oder wählen Sie mit ↵ einen solchen aus der Eingabeaufzeichnungsliste. Im Gegensatz zu *Search/Find* findet dieses Kommando die Deklaration der Funktion bzw. Prozedur, nicht aber die Stellen, wo sie benutzt werden (Aufrufe).

Find error ALT F8

Mit *Search/Find error* kann man die Stelle im Programm finden, an der ein Laufzeitfehler aufgetreten ist. Innerhalb der integrierten Entwicklungsumgebung werden Laufzeitfehler automatisch lokalisiert, falls das Programm mit Debug-Informationen (Schalter in *Options/Debugger* ist aktiviert) übersetzt worden ist. Auch wird die Fehleradresse als voreingestellter Wert gespeichert, so daß nach einem Dateiwechsel die Fehlerstelle wieder gefunden werden kann. Inner-

halb derselben Datei kann die letzte Laufzeitfehlermeldung und Fehlerposition durch Ctrl Q W jederzeit ermittelt werden.

Tritt ein Laufzeitfehler auf, während das Programm unter DOS läuft, zeigt das Laufzeitsystem die Fehleradresse an. Laden Sie in diesem Fall Turbo Pascal, und geben Sie die Programm-Datei als *Main file* im Compile-Menü an; setzen Sie *Destination* auf *Disk*. Dann tragen Sie im Menü *Search/Find error* die Fehleradresse ein. Die Fehleradresse muß im Format "xxxx:yyyy" in hexadezimaler Notation vorliegen, z.B. 2BE0:FFD4.

Hinweis: Sollte Turbo Pascal die Fehlerstelle nicht finden (Meldung: *Target address not found*), kann das verschiedene Gründe haben. Der Laufzeitfehler ist außerhalb des Programmcodes aufgetreten und liefert deshalb Werte die keiner Programmzeile zugeordnet sind. Dies kann zum Beispiel passieren, wenn der Laufzeitfehler in einer **external**-Routine auftritt, der natürlich keine Programmzeilen zugeordnet sind. In diesem Fall kann man zumindest mit der Map-Datei die Fehlerstelle eingrenzen. Ein weiterer Grund können völlig aberwitzige Werte sein (wie z.B. B00A:1234), die in Speicherbereichen jenseits des Bildwiederholspeichers liegen. Solche Werte entstehen durch Programme, die außer Kontrolle geraten und fälschlicherweise Teile des Programms und/oder Teile des Betriebssystems überschreiben.

Run-Menü

Alt R

Mit den Kommandos im *Run*-Menü können Sie Ihre Programme starten sowie Testphasen mit dem Debugger beginnen und beenden.

Run Ctrl F9

Run/Run startet Ihr Programm mit den Parametern, die unter *Run/Parameters* angegeben wurden. Sollte das Programm seit dem letzten Übersetzen verändert worden sein, führt die eingebaute Projektverwaltung ein Make durch und linkt anschließend das Programm.

Wollen Sie Ihr Programm in der integrierten Entwicklungsumgebung debuggen, muß der Schalter *Integrated* in *Options/Debugger* aktiviert sein. In diesem Fall wird spezielle Debug-Information erzeugt, die die Wirkung des *Run/Run*-Befehls wie folgt beeinflußt:

- Wenn das Programm seit der letzten Übersetzung nicht verändert wurde, startet *Run/Run* Ihr Programm und läßt es bis zum nächsten Breakpoint laufen oder, falls keiner gesetzt wurde, bis zum Ende.

- Wenn das Programm seit der letzten Übersetzung verändert wurde:
  - Falls Sie sich gerade in einer Debug-Sitzung befinden und *Run/Trace* oder *Run/Step* benützen, werden Sie von der Projektverwaltung gefragt, ob neu übersetzt werden soll.
  - Falls Sie *Yes* wählen, führt die Projektverwaltung ein Make durch.
  - Falls Sie *No* wählen, läuft das Programm weiter bis zum nächsten Breakpoint

bzw. zur nächsten Programmzeile oder bis zum Programmende, falls kein Breakpoint gesetzt war.

∩ Falls Sie sich nicht in einer Debug-Sitzung befinden, wird ein Make durchgeführt, und das Programm wird gelinkt.

Drücken von Ctrl Break stoppt das Programm an der aktuellen Programmzeile. Falls Turbo Pascal der Adresse der Unterbrechung keine Programmzeile zuordnen kann, kann mit einem zweiten Ctrl Break das Programm terminiert und in die integrierte Entwicklungsumgebung zurückgekehrt werden.

Program reset Ctrl F2

Run/Program reset stoppt den Testlauf mit dem Debugger, gibt den zusätzlich belegten Speicher frei und schließt alle geöffneten Dateien.

Verwenden Sie das Kommando, wenn Sie beim Debuggen feststellen, daß zu wenig Speicher für den Wechsel in DOS vorhanden ist oder falls Sie die Debug-Sitzung von vorne beginnen wollen.

Go to cursor F4

Run/Go to cursor startet das Programm ab dem hervorgehobenen Startbalken im Code bis zur Zeile, an der der Cursor im aktiven Fenster steht. Befindet sich in der Zeile keine ausführbare Anweisung, erscheint eine Warnung. Die Funktion kann auch einen Testlauf mit dem Debugger initiieren.

Dieses Kommando setzt selbst keine permanenten Breakpoints, stoppt aber das Programm, wenn es in der Zeile vor dem Cursor einen Breakpoint findet. Wählen Sie dann nochmal *Go to cursor*.

Der Startbalken kann auf den Teil, der mit dem Debugger getestet werden soll, verschoben werden. Soll das Programm an einer bestimmten Stelle stoppen, setzten Sie dort einen Breakpoint.

Steht der Cursor an einer Zeile, die nicht ausgeführt wird, läuft das Programm zum nächsten Breakpoint oder (wenn nicht vorhanden) zum Ende. Ctrl Break stoppt das Programm.

Trace into F7

Wählen Sie das Kommando *Trace/Into*, führt das Programm die einzelnen Anweisungen schrittweise aus. Bei einem Funktions- oder Prozeduraufruf werden auch die Anweisungen innerhalb des Unterprogramms durchlaufen und nicht in einem Schritt (vgl. *Run/Step over*) ausgeführt. Wird ein Unterprogrammaufruf gefunden, der für den Debugger nicht zugänglich ist, stoppt *Trace into* die Ausführung am Beginn der nächsten Anweisung.

Wird ein Unterprogrammaufruf gefunden, der für den Debugger zugänglich ist, stoppt *Trace into* die Ausführung am Anfang des Unterprogramms. Durch nachfolgende *Trace into*- oder *Step over*-Kommandos werden die Anweisungen in dem Unterprogramm ausgeführt.

Verläßt der Debugger die Prozedur, nimmt er die Auswertung der Anweisung, die den Aufruf enthält wieder auf:

```
if Max <= Min then
  DoSomeThing;
```

In diesem Beispiel kann man nur dann die Funktionen *Min* bzw. *Max* schrittweise verfolgen, falls die zugehörige Unit mit *Debug-Info* übersetzt wurde (in *Options/Compiler*) und der Schalter *Integrated* in *Options/Debugger* bei der Übersetzung aktiviert war.

Step over F8
Dieser Befehl führt die nächste Anweisung aus. Es wird nicht in Unterprogramme verzweigt, selbst wenn Debug-Informationen dafür erzeugt wurden. Mit anderen Worten behandelt dieser Befehl Prozeduren und Funktionen als eine Anweisung, die als ganzes ausgeführt wird.

Parameters
Mit *Run/Parameters* können Sie Programmen beim Start Kommandozeilen-Argumente übergeben. DOS-Kommandos für die Umleitung der Ausgabe werden ignoriert.

Geben Sie in das Fenster (siehe Abbildung 7.9) nur die Parameter, nicht den Programmnamen ein. Parameter wirken sich nur aus, wenn das Programm gestartet wird. Nachträgliche Änderungen werden erst nach *Program Reset* übernommen.

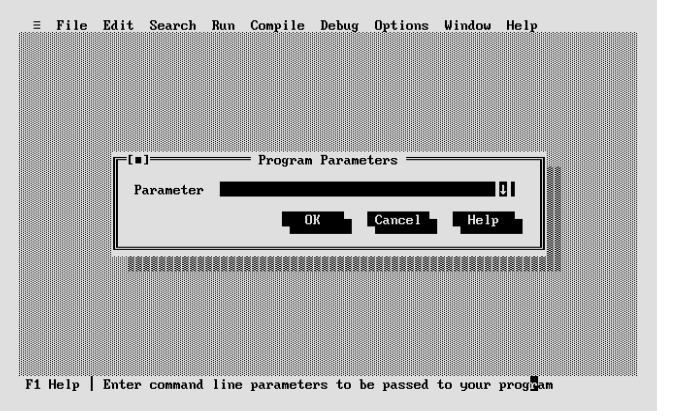


Abbildung 7.9 Das Dialogfenster Parameters

Compile-Menu

Alt C

compiliert das Programm im aktuellen Fenster oder führt für das Projekt ein Make oder Build durch. Wenn Sie die Kommandos *Compile*, *Make* und *Build* verwenden, muß in einem aktiven Editorfenster eine Datei geöffnet oder bei *Primary file* eine Datei definiert sein.

Compile Alt F9
übersetzt die Datei im aktuellen Fenster. Der Fortgang und die Ergebnisse der

Übersetzung werden in einem Status-Fenster angezeigt, das am Ende durch einen beliebigen Tastendruck wieder geschlossen werden kann. Dabei wird angezeigt, ob die Datei fehlerfrei übersetzt werden konnte oder ob ein Fehler aufgetreten ist. Der Übersetzungsprozeß kann mit Ctrl Break abgebrochen werden.

Make F9

Dieser Befehl führt ein Make durch, d.h. die aktuelle Datei (bzw. die Datei, die unter *Primary file* angegeben wurde) wird übersetzt. Dabei wird überprüft, ob verwendete Units veraltet sind, also der Quelltext jünger als die .TPU-Datei ist. Ist dies der Fall, werden diese Units auch übersetzt. Verwendet ein Programm (bzw. eine Unit) eine .OBJ-Datei, wird auch dann neu übersetzt, wenn die .OBJ-Datei jüngeren Datums ist als das Modul, das sie benutzt.

Achtung: Falls die Quelldatei nicht gefunden wird, verwendet Turbo Pascal die alte .TPU-Datei, ohne eine besondere Meldung auszugeben. Sollte Ihr Programm auf eine im Quelltext einer Unit vorgenommene Änderung nicht reagieren, könnte das der Grund dafür sein.

Build

Dieser Befehl übersetzt alle zum Projekt gehörenden Dateien neu. Zuerst werden alle Datums- und Zeiteinträge der Dateien auf Null gesetzt und anschließend ein Make durchgeführt.Wurde der Übersetzungsvorgang durch Drücken von Ctrl Break oder einem aufgetretenen Fehler unterbrochen, können Sie die Übersetzung mit *Compile/Make* einfach fortsetzen. Diese Vorgehensweise erspart die Neuübersetzung der bis dahin schon neu übersetzten Dateien.

Destination

Diese Option bestimmt, ob der erzeugte Code als .EXE-Datei auf Platte/Diskette geschrieben wird oder nur temporär im Hauptspeicher gehalten werden soll. Da eine .EXE-Datei jederzeit erzeugt werden kann, empfiehlt sich während der Entwicklungsphase das Übersetzen in den Hauptspeicher, das schneller ist, da weniger Plattenzugriffe notwendig sind. Die .EXE-Datei wird im Verzeichnis des Quelltextes abgespeichert oder in dem Verzeichnis, das in *Options/Directories/EXE & TPU directories* spezifiziert worden ist.

Primary file

Dieser Befehl spezifiziert die Datei, die übersetzt werden soll, falls *Compile/Make* oder *Compile/Build* gewählt wurde. Dies ist vorteilhaft, wenn Sie ein Programm haben, das aus mehreren Units besteht (das tut wohl fast jedes Programm) und gerade eine der Units editiert haben. Jetzt können Sie einfach mit F9 ein Make für das ganze Projekt durchführen. Ohne Angabe der Programm-Datei bei *Primary file* würde nur die Datei im aktuellen Fenster (und Dateien, von denen sie abhängt) übersetzt. Wollen Sie wirklich nur die Datei im aktuellen Editorfenster übersetzen, wählen Sie *Compile* (ALT F9).

Debug-Menü

Alt D

Mit diesem Kommando können Sie den Einsatz des integrierten Debuggers steuern. Voreinstellungen definieren Sie im Dialogfenster *Options/Debugger*.

Evaluate/modify Ctrl F4

Debug/Evaluate/modify berechnet eine Variable oder einen Ausdruck und zeigt den Wert an, den Sie ggf. auch ändern können. Das Dialogfenster (siehe Abbildung 7.10) enthält drei Felder: *Expression*, *Result* und *New value*.

Die Voreinstellung ist *Evaluate*. Im *Expression*-Feld finden Sie das Wort, auf dem der Cursor im Editorfenster steht. Diesen voreingestellten Ausdruck können Sie nach Drücken von Enter berechnen, editieren oder ändern. Mit ↓ erweitern Sie den vom Editorfenster übernommenen Text.

Kann der Debugger den Wert berechnen, erscheint er im Ergebnisfeld *Result*. Verweist der Ausdruck auf eine Variable oder auf ein einfaches Datenelement, können Sie im Feld *New value* einen neuen Wert eingeben. Esc schließt das Fenster. Neue Werte werden nur nach Wahl von *Modify* übernommen.

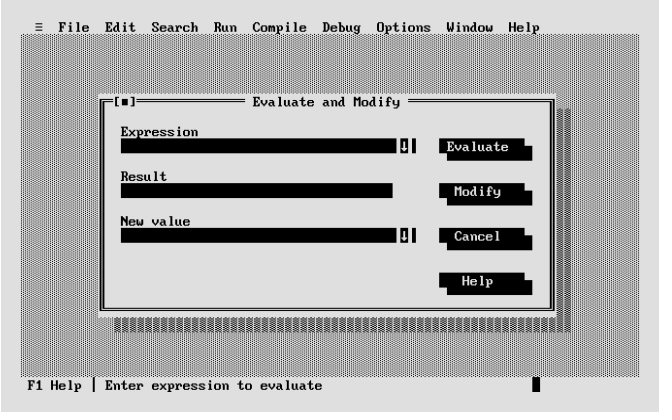


Abbildung 7.10 Das Dialogfenster Evaluate/modify

Wiederholungsausdrücke zeigen die Werte von Folgen aus Datenelementen an, z.B. für ein Integer-Array *xarray*:

xarray [0],5 zeigt 5 aufeinander folgende Integer-Werte dezimal an;

xarray [0],5x zeigt 5 aufeinander folgende Integer-Werte hexadezimal.

Ein Ausdruck, verwendet mit einem Wiederholungszähler, muß ein einzelnes Datenelement repräsentieren.

Debug/Evaluate/modify zeigt jeden Typ eines Wertes im geeigneten Format an. Die Anzeigeformate stellen Sie mit einem Komma, gefolgt von einem Formatbezeichner aus Tabelle 7.1 ein.

Tabelle 7.1 Formatangaben in Debugger-Ausdrücken

Zeichen	Funktion
C	»Zeichen« – Steuerzeichen (ASCII-Codes 0 bis 31) werden im Format #xx wiedergegeben. Anwendbar auf Zeichen und Strings.
S	»String« – Steuerzeichen (ASCII-Codes 0 bis 31) werden im Format #xx dargestellt. Da dieses Format der Standardvorgabe für einzelne Zeichen und Strings entspricht, ist die explizite Formatangabe **S** nur zusammen mit *M* sinnvoll.
D	»Dezimal« – Ganzzahlige Werte werden in dezimaler Form dargestellt. Anwendbar auf Integer-Ausdrücke, Arrays und Records mit Integer-Elementen.
$, H *oder* X	»Hexadezimal« – Ganzzahlige Werte werden in hexadezimaler Form mit einem vorangestellten $ dargestellt. Anwendbar auf Integerausdrücke, Zeichen, Arrays und Records mit Integer-Elementen.
Fn	»Gleitkommazahl« – *n* steht für eine ganze Zahl im Bereich von 2 bis 18 und legt die Anzahl der Nachkommastellen bei der Ausgabe fest (Vorgabewert ist 11). Nur auf Gleitkommazahlen anwendbar.
M	»Memory dump« – Speicherauszüge ab der durch den Ausdruck angegebenen Adresse. Der Ausdruck muß auf der linken Seite einer Zuweisung gültig sein, d.h. eine Speicheradresse ergeben; ansonsten wird die Formatangabe *M* ignoriert. Vorgabegemäß werden die gelesenen Werte im Hexadezimalformat (ohne vorangestelltes $) ausgegeben. In Kombination mit *D* (Ausdruck, *MD*) gibt Turbo Pascal die gelesenen Werte in dezimaler Form aus, die Kombination von *M* mit *X* erzeugt hexadezimale Ausgaben mit vorangestelltem $. Bei Anfügen von *C* oder *S* gibt Turbo Pascal die Werte in der Form *Zeichen* bzw. *String* aus, d.h. stellt ASCII-Zeichen dar. Die Anzahl der ausgegebenen Bytes wird standardmäßig durch die Größe der als Ausdruck angegebenen Variablen festgelegt, sie läßt sich durch einen zusätzlichen Wiederholungszähler ebenfalls explizit festlegen.
P	»Zeiger« – Ausgabe der Adresse, auf die eine Zeigervariable zeigt. Turbo Pascal verwendet hier die Form *Segment:Offset* anstelle des üblichen Formates *Zeiger*(*Segment:Offset*). Nur auf Zeiger (Pointer) anwendbar.
R	»Record/Variante« – Ausgabe einzelner Record- und Feldnamen (Namen und Werte), deren Typ durch den Ausdruck (wie z.B. { *X:1, Y:10, Z:5* }) festgelegt wird. Nur auf Records und Varianten anwendbar.

Watches

Debug/Watches blendet ein Pop-Up-Menü mit Kommandos ein, die die Verwendung von Breakpoints und Watch-Ausdrücken steuern.

Add Watch Ctrl F7 fügt einen Watch-Ausdruck in das Watch-Fenster ein. Der Debugger öffnet ein Fenster und fordert Sie auf, einen Watch-Ausdruck einzugeben. Voreingestellt ist das Wort, auf dem sich der Cursor im Editorfenster befindet. Aus der Eingabeaufzeichnungsliste können Sie frühere Eintragungen übernehmen.

Wenn Sie einen gültigen Ausdruck und Enter eingeben oder *OK* anklicken, fügt der Debugger den Ausdruck und den aktuellen Wert in das Watch-Fenster

ein. Ist das Watch-Fenster aktiv, können Sie einen neuen Watch-Ausdruck auch mit Ins einfügen.

Delete Watch löscht den aktuellen Watch-Ausdruck, wenn das Watch-Fenster aktiv ist. Der aktuelle Watch-Ausdruck wird, wenn das Watch-Fenster nicht aktiv ist, mit einem Punkt auf der linken Seite markiert. Einen so markierten Ausdruck können Sie mit *Delete/Watch* löschen. Andere Watch-Ausdrücke (ohne Punkt) löschen Sie nach Aktivieren des Watch-Fensters mit Del oder Ctrl Y .

Edit Watch editiert den aktuellen Watch-Ausdruck aus dem Watch-Fenster. Auch hier steht Ihnen eine Eingabeaufzeichnungsliste zu Verfügung.

Nach Anwahl öffnet der Debugger ein Dialogfenster mit dem aktuellen Watch-Ausdruck. Editieren Sie den Ausdruck, und drücken Sie Enter . Der Debugger ersetzt die ursprüngliche Version des Ausdrucks durch die editierte. Wenn das Watch-Fensters aktiv ist, können Sie einen Watch-Ausdruck auch sofort editieren, indem Sie ihn anwählen und Enter drücken.

Remove All Watches löscht alle Watch-Ausdrücke aus dem Watch-Fenster.

Toggle Breakpoint Ctrl F8

Debug/Toggle Breakpoint setzt oder löscht unbedingte Breakpoints an der Cursorposition. Quelltextzeilen, die einen Breakpoint enthalten, werden mit einem speziellen Zeichenattribut hervorgehoben.

Breakpoints

Debug/Breakpoints öffnet ein Fenster, das die Verwendung von bedingten und unbedingten Breakpoints steuert.

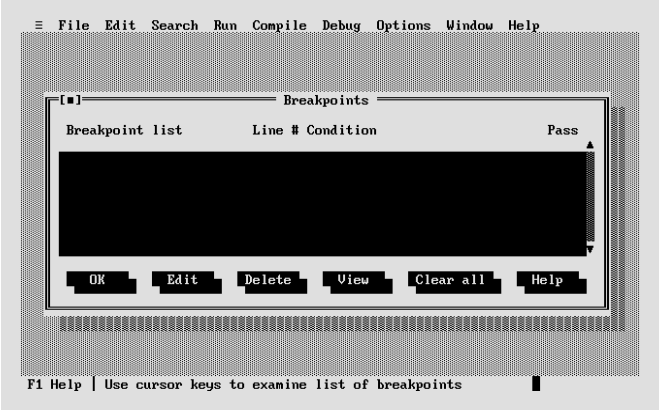


Abbildung 7.11 Das Dialogfenster Breakpoints

Das Dialogfenster zeigt alle gesetzten Breakpoints sowie deren Zeilennummern und die Bedingungen. Früher definierte Bedingungen können Sie auch aus der Eingabeaufzeichnungsliste übernehmen.

Der *Delete*-Schalter löscht Breakpoints aus dem Programm. Der Schalter *View* stellt den Cursor auf den aktiven Breakpoint im Quelltext, ohne dabei das Programm zu starten.

Edit fügt einen neuen Breakpoint in die Liste ein und öffnet dabei ein Dialogfenster (siehe Abbildung 7.12). Zeilennummer und Bedingungen des gesetzten Breakpoints werden angezeigt.

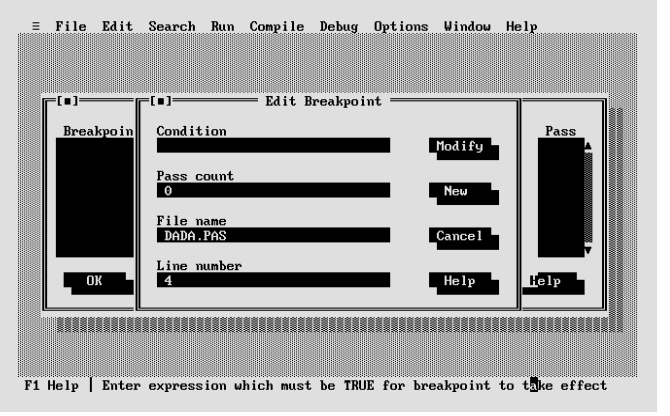


Abbildung 7.12
Das Dialogfenster Edit Breakpoint

Mit *Pass Count* geben Sie an, wie oft der Breakpoint übersprungen werden soll, bevor gestoppt wird. Mit dem *At*-Schalter können Sie einen Breakpoint einer bestimmten Funktion zuordnen (diese Option steht nur während des Testens mit dem Debugger zur Verfügung).

Mit *New* legen Sie Bedingungen für einen neuen Breakpoint fest, *Modify* übernimmt die veränderten Einstellungen für einen vorher definierten Breakpoint.

Das Programm stoppt immer, wenn es während des Programmlaufs auf einen Breakpoint stößt. Beim Programmstopp befindet sich der Startbalken in der Zeile, in der der Breakpoint steht. (Die Hervorhebung des Breakpoints wird durch die des Startbalkens verdeckt und erscheint wieder, wenn der Startbalken weiter bewegt wird.)

Wird eine Quelltextdatei editiert, bleibt jeder Breakpoint an der Zeile stehen, in der er gesetzt wurde. Breakpoints gehen nur verloren, wenn Sie

- die integrierte Entwicklungsumgebung verlassen,
- die Quelltextzeile, in der der Breakpoint gesetzt wurde, löschen,
- den Breakpoint mit *Toggle Breakpoint* löschen.

Turbo Pascal versucht in zwei Fällen, den Verlust von gesetzten Breakpoints zu verhindern:

- Wenn Sie eine Datei mit Breakpoints editieren und versuchen, die editierte Version der Datei nicht zu speichern.
- Wenn Sie eine Datei mit Breakpoints editieren und dann den aktuellen Debugger-Lauf fortsetzen, ohne einen neuen Make durchzuführen (Turbo Pascal warnt dann mit der Frage *Source modified, rebuild*?).

Bevor Sie einen Quelltext compilieren, können Sie in jeder beliebigen Zeile Breakpoints setzen, auch in einer Leerzeile oder in einem Kommentar. Wenn Sie die Datei compilieren und starten, prüft Turbo Pascal zuvor die Gültigkeit jedes gesetzten Breakpoints und gibt Ihnen die Möglichkeit, ungültige Breakpoints zu verschieben, zu ignorieren oder zu ändern. Wenn Sie die Datei mit dem Debugger testen, erkennt Turbo Pascal die Zeilen mit ausführbaren Anweisungen und warnt Sie, wenn Sie versuchen, einen ungültigen Breakpoint zu setzen.

Options-Menü

Alt O

Im *Options*-Menü finden Sie Kommandos, mit denen Sie viele Voreinstellungen von Turbo Pascal ansehen und ändern können. In der Regel wird hierzu ein Dialogfenster geöffnet.

Compiler

Options/Compiler öffnet ein Dialogfenster mit verschiedenen Optionen zur Beeinflussung der Code-Generierung.

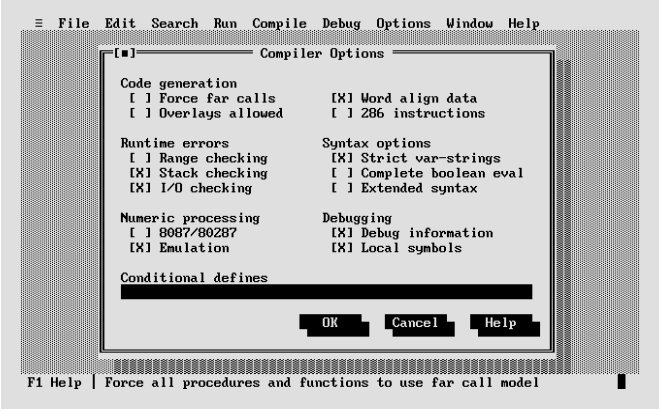


Abbildung 7.13 Das Dialogfenster Compiler Options

Code generation bestimmt verschiedene Optionen bei der Code-Erzeugung.

Force far calls weist den Compiler an, für Unterprogramme **far**-Aufrufe zu verwenden. Diese Option ist gleichbedeutend mit der Compiler-Direktive $F+ bzw. der FAR-Direktive für Unterprogramme, die sich allerdings nur auf das jeweilige Unterprogramm bezieht.

Overlays allowed steuert die Erzeugung von overlay-spezifischem Code. Dieser ist notwendig, um Strings und Mengen-Konstanten als Parameter übergeben zu können. Diese Option hat ihre Entsprechung in der Direktive $O+. Units, die mit *Overlays allowed* übersetzt worden sind, müssen nicht unbedingt als Over-

lays verwendet werden. Die Übersetzung stellt allerdings sicher, daß dies mit derselben Version der Unit möglich ist.

Word align data weist Turbo Pascal an, Datenstrukturen, die größer als ein Byte sind, an geraden Speicheradressen auszurichten. Dies bringt bei Prozessoren mit 16-Bit-Bus (8086, 80286, 80386, 80486) Geschwindigkeitsvorteile, da der Zugriff in einem Speicherzyklus geschehen kann.

286 instructions steuert die Erzeugung von 80286-spezifischen Befehlen. Auch diese Option bringt bei 80286-Prozessoren und deren Nachfolgern zum Teil deutliche Geschwindigkeitsvorteile. Programme, die mit dieser Option übersetzt worden sind, laufen natürlich nur noch auf Rechnern mit 80286 (386/486)-Prozessoren.

Runtime errors bestimmt, welche Laufzeitfehler erzeugt werden sollen.

Range checking erzeugt speziellen Code im Programm, der den Index bei Array (String)-Zugriffen auf Gültigkeit prüft. Sollte der Index außerhalb der definierten Grenzen liegen, erzeugt das Laufzeitsystem einen Fehler. Gleiches gilt für Zuweisungen an skalare Typen. Die Compiler-Direktive $R+ist zu dieser Option äquivalent.

Stack checking veranlaßt den Compiler, bei Unterprogrammaufrufen sicherzustellen, daß genügend Speicher auf dem Stack für lokale Variablen vorhanden ist. Andernfalls bricht das Programm mit einem Laufzeitfehler ab. $S+ ist die dieser Option entsprechende Compiler-Direktive.

I/O checking erzeugt Code, der nach jeder I/O-Operation den Fehlerstatus ermittelt. Wenn ein Fehler aufgetreten ist, terminiert das Programm. Ist die Option ausgeschaltet, wird kein Laufzeitfehler erzeugt. Der Status der I/O-Operation kann durch Abfrage der Variablen *IOResult* überprüft werden. Mit der Compiler-Direktive $I+erreicht man dasselbe.

Syntax options bestimmt die Verhaltensweise des Compilers bei der Syntaxprüfung.

Strict var-strings veranlaßt den Compiler, bei *Var-String*-Parametern zu prüfen, ob der Typ des formalen Parameters mit dem Typ des übergebenen Parameters übereinstimmt. Ist dieser Punkt nicht aktiviert, wird keine Überprüfung vorgenommen. Entsprechende Compiler-Direktive: $V+.

Complete boolean evaluation steuert die Auswertung von booleschen Ausdrücken. Wird diese Option aktiviert, wertet der Compiler sämtliche Teile eines Ausdrucks aus, selbst wenn bereits feststeht, daß sich am Gesamtergebnis durch Auswertung weiterer Teilausdrücke nichts mehr ändert. Entsprechende Compiler-Direktive: $B+.

Extended syntax ermöglicht die Verwendung einer syntaktischen Variante. Das heißt, daß eine Funktion wie eine Prozedur aufgerufen werden kann. Diese Option berührt allerdings nicht die Funktionen der Unit *System*.

Numeric processing steuert die Unterstützung der Gleitkomma-Arithmetik.

8087/80287 weist den Compiler an, 80x87-Code zu erzeugen. So übersetzte Programme laufen nur noch auf Rechnern mit Coprozessor. Entsprechende Compiler-Direktive: $N+.

Emulation führt zur Einbindung des 80x87-Emulators. In diesem Fall läuft das Programm sowohl auf Rechnern mit Coprozessor als auch auf solchen ohne Coprozessor. Entsprechende Compiler-Direktive $E+.

Memory Sizes

Diese Option entspricht der Compiler-Direktive $M. Sie erlaubt die Einstellung der Stack-Größe sowie der Minimal- und Maximal-Größe des Heaps.

Stack Size legt die Größe des Stacks fest. Der voreingestellte Wert ist 16 KByte.

Low Heap Limit legt die Minimal-Größe fest, die ein Programm an Heap-Speicher braucht. Die Voreinstellung ist 0 KByte. Ist beim Start des Programmes der verfügbare Heap kleiner als der eingestellte Wert, bricht das Programm mit einer Fehlermeldung ab.

High Heap Limit legt die Maximal-Größe des Heaps fest. Die Voreinstellung ist 655360 Bytes. Dieser Wert bestimmt die maximale Größe des Heaps. Ist weniger Speicherplatz verfügbar, wird soviel wie möglich belegt. Will man mit der Prozedur **Execute** andere Programme aufrufen, ist es wichtig, durch Einstellen von *High Heap Limit* den Speicherbedarf des Programmes, das ein anderes starten soll, zu begrenzen.

Linker

In diesem Dialogfenster lassen sich Einstellungen des integrierten Linkers von Turbo Pascal vornehmen.

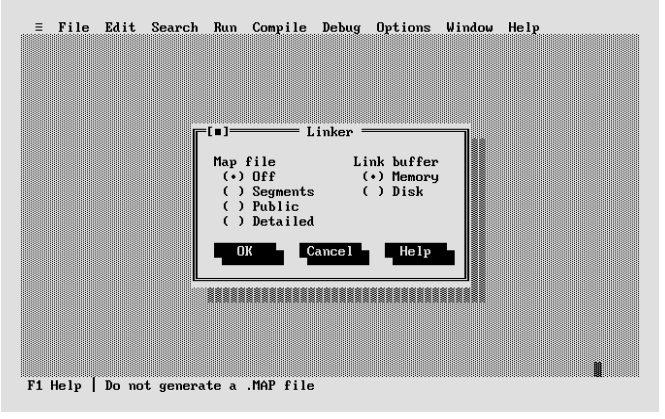


Abbildung 7.14 Das Dialogfenster Linker

Map File legt fest, ob der Linker eine Datei mit Angaben über Symbol-Adressen erzeugt und wenn ja, wie detailliert diese Angaben sind. Die drei Möglichkeiten *Segments* (Segmentadressen), *Publics* (öffentliche Symbole) und *Detailed* (sämtliche Daten) erzeugen eine Datei mit dem Suffix .MAP. Gemäß der Voreinstellung wird keine .MAP-Datei erzeugt.

Link Buffer legt fest, ob temporäre Daten, die während des Linkvorgangs erzeugt werden, im Hauptspeicher gehalten oder auf Platte zwischengespeichert werden sollen. Sollten Sie bei der Übersetzung Speicherplatzprobleme bekom-

men, können Sie durch die Option *Disk* einiges an Speicherplatz im Hauptspeicher sparen.

Debugger Dieses Dialogfenster betrifft Einstellungen des Debuggers (siehe Abbildung 7.15).

Abbildung 7.15 Das Dialogfenster Debugger

Debugging steuert die Erzeugung von Debug-Informationen.

Integrated – ist diese Option gesetzt, können Programme sowohl in der integrierten Entwicklungsumgebung als auch mit dem Turbo Debugger bearbeitet werden.

Falls *Standalone* gesetzt ist, kann das Programm nur noch mit dem Turbo Debugger bearbeitet werden. Die Option *Standalone* (in *Options/Debugger*) und *Map File* (in *Options/Linker*) erzeugt nur dann vollständige oder lokale Debug-Informationen, wenn das Modul mit der Option *Debug Information* (in *Options/Compiler*) bzw. *Local Symbols* (in *Options/Compiler*) übersetzt wurde.

Display Swapping Der Debugger wechselt zwischen Ausgabe-Fenster und IDE-Bildschirm, während das Programm läuft. *None* sollten Sie nur markieren, wenn Sie ganz sicher sind, daß beim Testen mit dem Debugger keine Ausgabe am Bildschirm erscheint.

Ist die Einstellung *Smart*, überprüft der Debugger, ob der Code eine Bildschirmausgabe erstellt. Wenn nicht (oder wenn er eine Funktion aufruft), wird die Ausgabe am Benutzerbildschirm angezeigt und dann wieder auf den IDE-Bildschirm umgeschaltet. Was Sie beachten sollten, wenn *Smart* markiert ist:

- Die Bildschirmanzeige wechselt bei jedem Funktionsaufruf, auch wenn die Funktion keine Bildschirmausgabe erzeugt.
- Manchmal ändert sich der IDE-Bildschirm auch ohne wechseln, z.B. wenn eine Timer-Interrupt-Routine auf den Bildschirm schreibt.

Ist *Always* markiert, wechselt der Debugger den Bildschirm bei jeder Anweisung, die ausgeführt wird. Diese Einstellung sollten Sie wählen, wenn der IDE-Bildschirm wahrscheinlich vom laufenden Programm überschrieben wird.

Anmerkung: Benutzen Sie den Debugger im Dual-Monitor-Modus (d.h. mit

der Kommandozeilen-Option /d), sehen Sie auf einem Bildschirm die Programmausgabe, auf dem anderen den Turbo-Pascal-Bildschirm. In diesem Fall ist ein Umschalten nicht nötig und die Einstellungen bleiben ohne Wirkung.

Directories

Mit *Options/Directories* bestimmen Sie, wo Turbo Pascal die Dateien zum Compilieren/Linken findet und wohin Ausgaben geschrieben werden sollen.

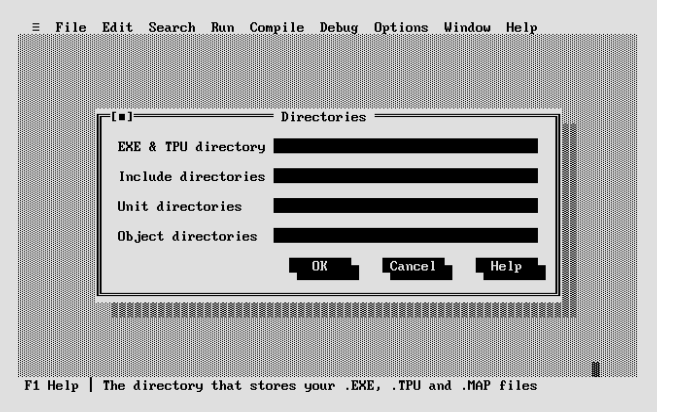


Abbildung 7.16 Das Dialogfenster Directories

Beachten Sie folgendes, wenn Sie Verzeichnisnamen angeben:

- Wenn die Angabe mehrerer Pfadnamen erlaubt ist, müssen Sie sie mit einem Semikolon voneinander trennen. Zulässig sind Namen bis zu 127 Zeichen (einschließlich Whitespace).
- Whitespace ist vor und nach einem Semikolon erlaubt, aber nicht nötig.
- Relative und absolute Pfadnamen sind erlaubt, eingeschlossen sind auch Pfadnamen zu anderen Laufwerken als dem aktuellen. Ein Beispiel:

```
C:\PASCAL;C:\PASCAL\MYPROJS;A:\TURBO\EXAMPLES;
```

Das Dialogfenster *Directories* enthält folgende Eingabefelder:

EXE & TPU directory legt fest, wohin .EXE- oder .TPU-Dateien geschrieben werden. Wenn kein Eintrag erfolgt, werden die Dateien in das Verzeichnis geschrieben, in dem sich die Quelldatei befindet. .MAP-Dateien werden ebenso behandelt, wenn *Map file* unter *Options/Linker* gesezt ist.

Include directories beschreibt, wo sich Ihre Standard-Include-Dateien befinden. Das sind die Dateien, die mit der Compiler-Direktive {$I *dateiname*} angegeben werden.

In *Unit directories* werden die Verzeichnisse spezifiziert, welche die Units enthalten.

Object directories beschreibt, wo die .OBJ-Dateien zu finden sind. Wenn Sie die Direktive {$L *dateiname*} angeben, sucht Turbo Pascal zunächst im aktuellen Verzeichnis, dann in demjenigen, das Sie über die Direktive spezifiziert haben.

Enviroment

Mit *Options/Enviroment* können Sie globale Einstellungen der integrierten Entwicklungsumgebung definieren. Drei Wahlpunkte *Preferences*, *Editor* und *Mouse* (bei *Full Menus*) werden Ihnen angeboten (siehe Abbildung 7.17).

Preferences/Screen Size Hier können Sie festlegen, ob der IDE-Bildschirm 25 oder 43/50 Zeilen darstellen soll. Je nachdem, welchen Video-Adapter Ihr PC benutzt, wird Ihnen einer oder beide Schalter angeboten.

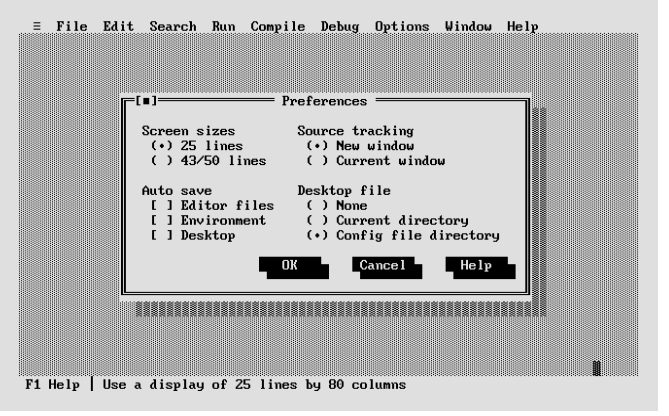


Abbildung 7.17 Das Dialogfenster Preferences

Übernehmen Sie die Voreinstellung, werden 25 Zeilen und 80 Spalten angezeigt. Dies ist die einzige Bildschirmdarstellung, die Systeme mit Monochrom-Anzeige und mit Farb-Graphik-Adaptern (CGA) erlauben. Arbeiten Sie mit EGA oder VGA, können Sie auf 43/50 Zeilen-Modus umschalten.

Preferences/Source Tracking Wenn Sie ein Programm schrittweise ablaufen lassen, öffnet die IDE – im Falle von *New Window* – immer dann ein neues Fenster, wenn es auf eine Datei stößt, die noch nicht geladen ist. Wählen Sie stattdessen *Current Window*, ersetzt die IDE den Inhalt des vordersten Fensters durch die neue Datei und öffnet kein neues Editorfenster.

Preferences/Auto Save *Editor Files* speichert automatisch, wenn Sie *Run/Run* oder *File/DOS Shell* (oder ein *Debug/Run*-Kommando) wählen, die Datei aus dem Editorfenster, wenn sie seit dem letzten Abspeichern verändert wurde.

Ist *Environment* markiert, werden alle Einstellungen aus dem gleichnamigen Dialogfenster automatisch gespeichert, wenn Sie Turbo Pascal verlassen.

Ist *Desktop* markiert, prüft Turbo Pascal, ob alle Dateien der Arbeitsfläche beim Verlassen gesichert sind und wieder geladen werden können, wenn Sie in Turbo Pascal zurückkehren.

Environment/Editor Im *Editor/Options*-Dialogfenster finden Sie die folgenden Optionen für den Umgang mit Texten:

Create Backup Files Turbo Pascal erstellt immer, wenn Sie *File Save* wählen, von der Quell-Datei, die sich im Editorfenster befindet, automatisch ein Backup mit der Endung .BAK. Dies ist die Voreinstellung.

Insert Mode Der eingegebene Text wird eingefügt. Im Editorfenster können Sie den Modus mit Ins ausschalten, d.h. eingegebener Text überschreibt vorhandenen.

Autoindent Mode Drücken Sie im Editorfenster Enter , springt der Cursor auf die Spaltenposition des ersten Zeichens der vorhergehenden nicht leeren Zeile. Dieser automatische Einzug ist sehr hilfreich beim Strukturieren des Codes.

Use tab character Turbo Pascal fügt ein echtes Tabulator-Zeichen (ASCII 9) ein, wenn Sie Tab drücken. Ist die Option nicht markiert, werden die Tabulatoren durch Leerzeichen ersetzt.

Optimal fill Turbo Pascal füllt jede automatisch eingerückte Zeile mit der minimalen Anzahl von möglichen Zeichen und verwendet, wenn nötig, Tabulatoren und Leerstellen. Dadurch enthalten die Zeilen sowenig Zeichen wie möglich.

Backspace unindents Steht der Cursor in einer Leerzeile oder auf dem ersten nicht leeren Zeichen einer Zeile, richtet Backspace die Zeile nach der vorherigen Einrückung aus.

Cursor through tabs Die Pfeiltasten positionieren den Cursor in die Mitte der Tabulatoren; ansonsten überspringt der Cursor mehrere Spalten, wenn er über einen Tabulator bewegt wird.

Tab Size 8 Ist *Use Tab Character* markiert, und Sie drücken Tab , wird ein Tabulator-Zeichen eingefügt; der Cursor springt zum nächsten Tabulator. Geben Sie im *Tab-Size*-Fenster an, wie groß der Abstand zwischen den Tabulatoren sein soll. Die Voreinstellung ist 8 Zeichen, möglich sind Werte zwischen 2 und 16.

Ändern Sie hier den Tabulatorabstand, werden die Tabulatorpositionen in Ihren Dateien sofort entsprechend angezeigt. Mit *Options/Save* speichern Sie den neuen Tabulatorabstand.

Environment/Mouse Dieses Kommando öffnet ein Dialogfenster mit Einstellungen für die Maus. Verfügbare Optionen:

Mit Right Mouse Button stellen Sie ein, welche Auswirkungen der Druck auf die rechte Maustaste (oder auf die linke, wenn die Option *Reverse Mouse Button* markiert ist) hat. Die Voreinstellung ist *Topic Search*. Alternative Einstellungen, abgesehen von *Nothing*, sind:

Topic Search	entspricht	*Help/Topic Search*
Go to Cursor	entspricht	*Run/Go To Cursor*
Breakpoint	entspricht	*Debug/Toggle Breakpoint*
Evaluate	entspricht	*Debug/Evaluate*
Add Watch	entspricht	*Debug/Watches/Add Watch*

Im Fenster *Mouse Double Click* können Sie die Geschwindigkeit bzw. Verzögerung des Doppelklicks der Maustasten mit den Cursortasten einstellen.

Bewegen Sie den Gleitregler nach *Fast*, braucht Turbo Pascal weniger Zeit, um einen doppelten von einem einfachen Klick zu unterscheiden. Bewegen Sie ihn zu *Slow*, erkennt Turbo Pascal doppeltes Anklicken auch, wenn Sie sich für das erneute Drücken der Maustaste mehr Zeit lassen.

Environment/Startup Über *Startup* können Sie die Einstellungen der integrierten Entwicklungsumgebung ändern (siehe Abbildung 7.18).

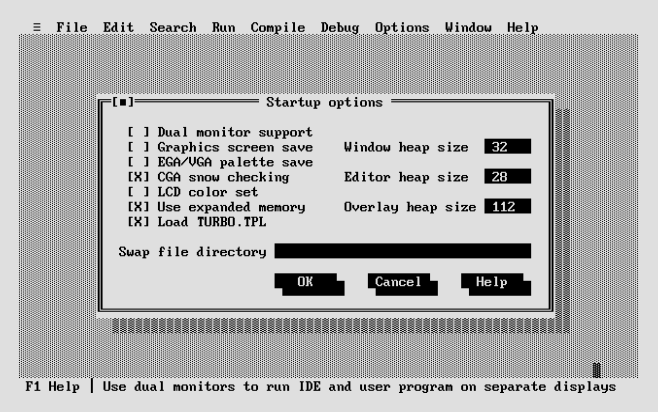


Abbildung 7.18 Das Dialogfenster Startup Options

Alle Optionen korrespondieren mit den Kommandozeilen-Optionen, die zu Beginn dieses Kapitels beschrieben worden sind.

Alle Änderungen, die Sie hier vornehmen, werden in die Datei TURBO.EXE geschrieben und erst beim nächsten Laden der IDE aktiviert.

Environment/Colors Mit *Colors* verändern Sie die Farbgebung der IDE (siehe Abbildung 7.19).

Abbildung 7.19 Das Dialogfenster Colors

Das Fenster *Group* zeigt eine Liste aller Bereiche, deren Farben Sie anpassen können. Wenn Sie einen Bereich auswählen, zeigt das Fenster *Item* alles, was in diesem Bereich geändert werden kann.

Wenn Sie ein System mit einem Farb- oder einem Schwarzweißbildschirm benutzen, wählen Sie die Vorder- und Hintergrundfarben mit der Maus oder den Cursortasten aus der Farbpalette aus. Bei allen Systemen zeigt der rechte untere Bildschirmbereich die aktuellen Einstellungen. Änderungen werden erst

wirksam, wenn Sie dieses Dialogfenster wieder verlassen (und *OK* ausgewählt haben).

Options/Save

Wählen Sie *Options/Save*, und sichern Sie die Einstellungen, die Sie in den Dialogfenstern von *Find* und *Replace* im Search-Menü, *Destination* und *Primary file* im Compile-Menü und im Options-Menü vorgenommen haben. Jede Kategorie wird in einer Datei gespeichert.

Werden die Dateien nicht gefunden, sucht Turbo Pascal in dem Verzeichnis, von dem aus TURBO.EXE gestartet wurde.

Options/Retrieve

Options/Retrieve zeigt ein Dialogfenster, über das Sie die Einstellungen, die Sie in den Dialogfenstern von *Find* und *Replace* im Search-Menü, *Destination* und *Primary file* im Compile-Menü und im Options-Menü vorgenommen haben, wieder laden können. Wenn *Desktop file* (.DSK) entweder auf *Current directory* oder *Config file directory* gesetzt ist, wird auch TURBO.DSK geladen.

Window-Menü

Alt W

Im Window-Menü finden Sie die Befehle zur Verwaltung der Fenster. Die meisten Fenster enthalten die Standard-Fensterelemente wie Rollbalken sowie Schließ- und Zoom-Feld. *Window/List* zeigt alle offenen Fenster an.

Size/Move Ctrl F5

Mit *Window/Size/Move* ändern Sie Position und Größe des aktiven Fensters. Das aktive Fenster folgt den Pfeiltasten. Ist es an der richtigen Stelle, drücken Sie Enter. Sie können das Fenster auch durch Verschieben des Titelbalkens mit der Maus bewegen.

Möchten Sie auch die Größe verändern, drücken Sie während des Verschiebens mit den Pfeiltasten die Taste Shift. Ist die richtige Größe erreicht, drücken Sie Enter.

Finden Sie im Fenster ein Feld zur Größenänderung, dann können Sie die Größe durch Ziehen des Feldes ändern.

Zoom F5

Mit *Window/Zoom* vergrößern Sie das aktive Fenster bis zum Maximum. Wird die maximale Fenstergröße bereits angezeigt, stellt das Kommando die vorherige Größe wieder her.

Denselben Effekt erreichen Sie durch doppeltes Anklicken der obersten Zeile (außerhalb von Icons).

Tile

Mit *Window/Tile* werden alle offenen Editorfenster ohne Überlappung auf dem Bildschirm angezeigt.

Cascade
Mit *Window/Cascade* stapeln Sie alle offenen Editorfenster.

Next F6
Mit *Window/Next* aktivieren Sie das nächste Fenster.

Close Alt F3
Mit *Window/Close* oder durch Anklicken des Schließfeldes schließen Sie das aktive Fenster.

Watch
Window/Watch öffnet und aktiviert das Watch-Fenster. Es zeigt Ausdrücke und ihre wechselnden Werte, so daß Sie darauf achten können, wie sich Variableninhalte ändern.

Mit dem Kommando aus dem Menü *Debug/Watches*, entfernen Sie aus dem Fenster Watch-Ausdrücke oder fügen welche hinzu (vgl. auch den Abschnitt »Watches« in diesem Kapitel).

Schließen Sie das Fenster durch Anklicken des Schließfeldes oder mit *Window/Close*.

Output
Mit *Window/Output* öffnen und aktivieren Sie das Ausgabe-Fenster. Es zeigt Text von der DOS-Kommandozeile und vom Programm an (keine Grafiken).

Das Ausgabe-Fenster ist beim Testen der Programme mit dem Debugger verfügbar, damit Sie Ihren Quellcode, die Variablen und die Ausgabe insgesamt ansehen können. Das ist besonders vorteilhaft, wenn Sie im Dialogfenster *Options/Environment* 43/50 Zeilen eingestellt haben und ein Programm im Standardmodus mit 25 Zeilen laufen lassen. In diesem Fall sehen Sie die ganze Programmausgabe und haben immer noch genügend Zeilen zur Anzeige von Quellcode und Variablen zur Verfügung.

Soll der Programmtext auf dem ganzen Bildschirm erscheinen oder erstellt Ihr Programm Grafiken, wählen Sie *Window/User Screen*, um die Ausgabe zu betrachten.

Schließen Sie das Fenster durch Anklicken des Schließfeldes oder mit *Window/Close*.

Call Stack Ctrl F3
Window/Call Stack öffnet ein Fenster, das alle Prozeduren anzeigt, die Ihr Programm aufgerufen hat, bis diejenige erreicht worden ist, die gerade abläuft. An unterster Stelle steht der Programmname, an oberster Stelle die aktuelle Prozedur. Jeder Eintrag umfaßt den Namen der aufgerufenen Prozedur und die Werte aller Parameter, die an sie übergeben worden sind.

Der oberste Eintrag ist hervorgehoben dargestellt. Wenn Sie sich die Einträge zu einer anderen Prozedur anschauen möchten, geben Sie deren Namen an; schließen Sie die Eingabe mit Enter ab. Der Cursor bewegt sich dann zu der entsprechenden Information. Das Fenster *Call Stack* bleibt auf dem Bildschirm, bis Sie es schließen.

User Screen Alt F5

Window/User Screen zeigt die Programmausgabe am ganzen Bildschirm an. Mit *Window/Output* wird sie in einem Turbo-Pascal-Fenster angezeigt.

In die integrierte Entwicklungsumgebung gelangen Sie durch jeden Tastendruck.

Register

(Nur bei *Full menus*) *Window/Registers* öffnet und aktiviert das *Register*-Fenster.

Es zeigt die aktuellen Werte der CPU-Register und wird verwendet, wenn Sie mit dem integrierten Assembler arbeiten.

Schließen Sie das Fenster durch Anklicken des Schließfeldes oder mit *Window/Close*.

List

Mit *Window/List* erhalten Sie die Namen aller aktuell geöffneten Fenster.

Alt 0

blendet eine Liste aller offenen Editorfenster ein (vgl. auch den Abschnitt über Fensterverwaltung in diesem Kapitel).

Help-Menü

F1

Das Help-Menü öffnet ein eigenes Fenster, über das Sie auf das Turbo-Pascal-Hilfesystem zugreifen können. Die Online-Hilfe enthält Informationen zu praktisch allen Punkten der integrierten Entwicklungsumgebung und Turbo Pascal. (Daneben erscheinen Hinweise auf das gewählte Menü-Kommando und verfügbare Tastenkürzel in der Statuszeile.)

Arbeitsschritte zum Öffnen eines Help-Fensters:

- Drücken Sie an einer beliebigen Stelle im Programm F1.
- Befindet sich der Cursor in einem Editorfenster und auf einem Wort, erhalten Sie mit Ctrl F1 sprachbezogene Hilfeinformationen.
- Erscheint *Help* in einer Statuszeile oder in einem Dialogfenster, können Sie den Begriff auch anklicken.

Schließen Sie das Help-Fenster mit Esc, durch Anklicken des Schließfeldes oder wählen Sie *Window/Close*. Das Help-Fenster kann, auch wenn Sie in einem anderen Fenster arbeiten, offen bleiben, wenn Sie es aus einem Dialogfenster oder mit F1, während ein Menü-Kommando gewählt war, geöffnet haben. Klicken Sie vom Help-Fenster aus ein anderes Fenster an oder drücken Sie F6, um ein anderes Fenster zu aktivieren.

Hilfebildschirme enthalten oft *Schlüsselwörter* (hervorgehobene Wörter), die Sie anwählen können, wenn Sie an weiteren Informationen interessiert sind. Tab bringt Sie zu einem Schlüsselwort, Enter zeigt die detaillierte Hilfe an.

Alternativ können Sie auch den Cursor auf das hervorgehobene Wort positionieren und Enter drücken. Mit Hilfe der Maus laden Sie den Hilfetext durch doppeltes Anklicken eines Schlüsselworts.

Weitere Hilfe erhalten Sie, indem Sie den Cursor im Help-Fenster auf ein *beliebiges* Wort positionieren und Ctrl F1 drücken. Wird das Wort nicht im Index gefunden, wird eine erweiterte Suche im Index durchgeführt und der Indexeintrag angezeigt, der dem Wort am ehesten entspricht.

Den Text aus einem aktiven Help-Fenster können Sie kopieren und in das Editorfenster einfügen. Die Verfahrensweise ist die gleiche wie beim Kopieren im Editorfenster: markieren Sie den Text mit Shift und den Cursortasten oder mit der Maus Wählen Sie *Edit/Copy*, wechseln Sie ins Editorfenster, und wählen Sie dann *Edit/Paste*.

Hinweis: Erhalten Sie in einem Dialogfenster oder in einem Menü Hilfe, dann können Sie das Fenster nicht verkleinern oder in die Zwischenablage kopieren. Mit Tab gelangen Sie in diesem Fall in die Dialogkontrolle des Fensters, nicht zum nächsten Schlüsselwort.

Die bereits markierten Programmbeispiele aus den Help-Fenstern kopieren Sie mit *Edit/Copy Example*.

Contents

Help/Contents öffnet das Help-Fenster und zeigt das Hauptverzeichnis an. Über dieses Fenster gelangen Sie in jeden Teil des Hilfesystems.

F1

Hilfe über das Hilfesystem erhalten Sie bei einem aktiven Help-Fenster mit F1 oder durch Anklicken der Statuszeile.

Index

Help/Index öffnet ein Dialogfenster mit einer vollständigen Liste von Hilfe-Schlüsselwörtern.

Die Liste können Sie rollen. Die Suche erweitern Sie durch Eingabe von Buchstaben über die Tastatur. Interessiert Sie z.B., was unter *printing* zu finden ist, geben Sie *pri* ein. Tippen Sie *p*, springt der Cursor zum ersten Schlüsselwort, das mit *p* beginnt. Bei Eingabe von *r* wird das erste Schlüsselwort mit *pr* angezeigt usw.

Schlüsselwörter wählen Sie mit Tab oder indem Sie den Cursor darauf positionieren und Enter drücken oder durch doppeltes Anklicken des Begriffs.

Topic Search Ctrl F1

Help/Topic Search zeigt sprachbezogene Hilfetexte zum aktuellen Element im Quelltext an.

Positionieren Sie den Cursor auf ein Element im Editorfenster, und wählen Sie *Topic Search*. Sie erhalten Hilfe über Funktionen (z.B. *writeln*), Units, reservierte Wörter usw. Kann das Element nicht im Hilfeindex gefunden werden, wird ein Indexeintrag angezeigt, der dem gesuchten Begriff am nächsten kommt.

Previous Topic Alt F1

Help/Previous Topic öffnet das Help-Fenster mit dem zuletzt angezeigten Hilfetext.

Insgesamt können Sie die letzten 20 Hilfebildschirme mit Alt F1 zurückgehen. Den letzten Hilfebildschirm erhalten Sie auch durch Anklicken der Statuszeile.

Help on Help F1

Help/Help öffnet einen Textbildschirm, in dem der Umgang mit dem Hilfesystem erklärt wird. Vom Help-Menü aus öffnen Sie den Bildschirm mit F1.

8. Der Editor von A bis Z

Dieses Kapitel beschreibt die Befehle des Editors von Turbo Pascal. Sie sollten dieses Kapitel auch dann lesen, wenn Sie schon mit den Editoren anderer Borland-Produkte vertraut sind, denn in der neuen integrierten Entwicklungsumgebung wurde der Editor weiter verbessert. Kontextbezogene Hilfe erhalten Sie jeweils mit F1. Eine Übersicht über alle Editor-Befehle finden Sie in Tabelle 8.1.

Eine Einführung in den Editor und die neue IDE von Turbo Pascal finden Sie in Kapitel 1 »Die neue integrierte Entwicklungsumgebung«, eine ausführliche Beschreibung in Kapitel 7 »Die IDE-Referenz«.

Der alte und der neue Editor

Die bereits aus anderen Borland-Produkten bekannten Tastenkürzel für Cursorbewegungen sowie zum Einfügen, Kopieren, Löschen, Suchen oder Ersetzen von Text haben sich nicht verändert. Hinzugekommen sind im Hauptmenü des neuen Editors ein *Edit*- und ein *Search*-Menü sowie Mausunterstützung für viele Cursorbewegungen und für Kommandos zum Markieren von Blöcken.

Das *Edit*-Menü enthält Befehle zum Ausschneiden, Kopieren und Einfügen von Dateien, Kopieren von Beispielen aus dem Hilfe-Menü in ein Editorfenster und zur Arbeit mit der Zwischenablage. Sie öffnen ein Editorfenster über das Menü *File* mit dem Kommando *Open*. Vom Editorfenster aus gelangen Sie mit F10 in das Auswahlmenü, zurück kommen Sie wieder mit Esc (unter der Voraussetzung, daß kein Untermenü geöffnet ist); arbeiten Sie mit der Maus, so reicht einfaches Klicken an einer beliebigen Stelle innerhalb des Fensters.

Die Eingabe von Text im Editor ist der Arbeit mit einer Schreibmaschine sehr ähnlich. Mit Enter schließen Sie eine Zeile ab. Ist der Bildschirm vollgeschrieben, wird die oberste Zeile aus dem Fenster geschoben. Keine Angst – sie geht dabei nicht verloren. Sie können mit Hilfe der Cursor-Bewegungsbefehle, die weiter unten in diesem Kapitel beschrieben werden, den Text vor- und zurückblättern.

Mit dem Kommando *Edit/Restore line* können Sie Veränderungen, die Sie in der zuletzt bearbeiteten Textzeile vorgenommen haben, wieder rückgängig machen.

Editor-Referenz

Es können hier nicht alle Funktionen des inzwischen sehr mächtigen Editors angesprochen werden. Zusätzlich zu den Menü-*Auswahlen* kennt der Editor insgesamt rund 50 verschiedene Kommandos zur Bewegung des Cursors, dem »Blättern« von Text, dem Suchen und Ersetzen von Textteilen usw. Die Kommandos lassen sich in vier verschiedene Gruppen einteilen:

- Bewegungen des Cursors
- Einfügen und Löschen
- Operationen mit Textblöcken
- Verschiedenes

Die meisten dieser Befehle bedürfen keiner weiteren Erklärung.

Hinweis: Ein Wort ist definiert als Zeichenfolge, getrennt durch eines der nachfolgenden Zeichen: Leerzeichen `< > . ; / ( ) { } ^ ' * + - / $ # _ = | ~ ? ! " % & ` : @ [ ] \` Kontroll- und Sonderzeichen.

Übersicht der Editor-Kommandos

Tabelle 8.1 Grundlegende Kommandos

Cursorbewegung	Kommando
Zeichen nach links	Ctrl S oder ↑
Zeichen nach rechts	Ctrl D oder ↓
Wort nach links	Ctrl A oder Ctrl ↑
Wort nach rechts	Ctrl F oder Ctrl ↓
Zeile nach oben	Ctrl E oder ←
Zeile nach unten	Ctrl X oder →
Aufwärts rollen	Ctrl W
Abwärts rollen	Ctrl Z
Seite nach oben	Ctrl R oder PgUp
Seite nach unten	Ctrl C oder PgDn

Viele der hier genannten Kommandos können Sie auch mit der Maus ausführen (siehe Kapitel 1 »Die neue integrierte Entwicklungsumgebung«).

Tabelle 8.2 Erweiterte Cursorbewegungen

Cursorbewegung	Kommando
Zeilenanfang	Home
Zeilenende	End
Fensterobergrenze	Ctrl Home
Fensteruntergrenze	Ctrl End

Cursorbewegung	Kommando
Dateianfang	Ctrl PgUp
Dateiende	Ctrl PgDn
Blockanfang	Ctrl Q B
Blockende	Ctrl Q K
Letzte Cursorposition	Ctrl Q P

Tabelle 8.3 Einfüge- und Lösch-Kommandos

Cursorbewegung	Kommando
Einfügemodus ein/aus	Ins oder *Options/Environment/Editor/Insert mode*
Zeichen links vom Cursor löschen	Backspace
Zeichen an Cursorposition löschen	Del
Wort rechts löschen	Ctrl T
Zeile einfügen	Ctrl N
Zeile löschen	Ctrl Y
Rest der Zeile löschen	Ctrl Q Y

Tabelle 8.4 Block-Kommandos

Cursorbewegung	Kommando
Block markieren	Shift →, Shift ←, Shift ↓, Shift ↑, Ctrl K B, Ctrl K K
Wort markieren	Ctrl K T
Block kopieren	*Edit/Copy*, *Edit/Paste* oder Ctrl Ins, Shift Ins
Block verlagern	*Edit/Cut*, *Edit/Paste* oder Shift Del, Shift Ins
Block löschen	*Edit/Clear* oder Ctrl Del
Block einlesen	Ctrl K R
Block speichern	Ctrl K W
Blockmarkierung ein/aus	Ctrl K H
Block drucken	*File/Print* oder Ctrl K P
Block einrücken	Ctrl K I
Block ausrücken	Ctrl K U

Tabelle 8.5 Weitere Editor-Kommandos

Cursorbewegung	Kommando
Auto-Einzug ein/aus	*Options/Environment/Editor/Autoindent mode**
Einleitung für Kontrollzeichen**	Ctrl P
Marke suchen	Ctrl Q n (n = Zifferntaste zwischen 0 bis 9)
Hauptmenü wählen	F10
Neue Datei	*File/New*
Datei laden	*File/Open* oder F3
Optimaler Füll-Modus ein/aus	*Options/Environment/Editor/Optimal fill**
Begrenzerpaare suchen	Ctrl, Ctrl Q [und Ctrl Q]
Datei drucken	*File/Print*

Cursorbewegung	Kommando
IDE verlassen	Alt X oder *File/Exit*
Letzte Suche wiederholen	Ctrl L oder *Search/Search again*
Fehlermeldung wieder ausgeben	Ctrl Q W
Zeile wiederherstellen	Ctrl Q L oder *Edit/Restore line*
Vom Menü zum Editor zurückkehren	Esc
Sichern	F2 oder *File/Save*
Suchen	Ctrl Q F oder *Search/Find*
Suchen und ersetzen	Ctrl Q F oder *Search/Replace*
Marke setzen	Ctrl K n (n = Zifferntaste zwischen 0 bis 9)
Tabulator	Tab
Tabulator-Modus	*Options/Environment/Editor/Use tab characters**
Ausrücken-Modus	*Options/Environment/Editor/Backspace unindents**

* Das Kommando öffnet das Dialogfenster Editor Options, in dem Sie die notwendigen Markierungsfelder oder Schalter einstellen können.
** Drücken Sie Ctrl R und drücken Sie dann das gewünschte Kontrollzeichen. Die Kontrollzeichen werden (abhängig von der Bildschirmeinstellung) in halber Helligkeit oder invers dargestellt.

Cursorsprünge

Drei Cursorbewegungen sollen näher beschrieben werden: Ctrl Q B (Cursor zum Blockanfang), Ctrl Q K (Cursor zum Blockende) und Ctrl Q P (Cursor zur letzten Cursorposition).

Ctrl Q B und Ctrl Q K bewegen den Cursor zum Anfang bzw. zum Ende der Blockmarkierung. Beide Befehle funktionieren auch dann, wenn die Blockmarkierung nicht angezeigt wird (siehe Blockmarkierung ein/aus in Tabelle 8.2) und wenn nur eine Markierung gesetzt ist.

Ctrl Q P stellt den Cursor auf die Position vor dem letzten Kommando. Nach einer Such- oder Suchen-/Ersetzen-Operation gelangen Sie damit wieder an die Cursorposition beim Start des Kommandos.

Blockkommandos

Ein Textblock umfaßt eine bestimmte Textmenge, die unterschiedlich groß sein kann (ein Zeichen bis zu Tausenden von Zeilen) und die speziellen Zeichen zur Blockmarkierung. In einem Fenster kann immer nur ein Block markiert sein. Ein Block wird markiert, indem auf dem ersten und nach letzten Zeichen des Textblocks eine Blockmarkierung angebracht wird. Markierte Blöcke können kopiert, verschoben, gelöscht, gedruckt oder in einer Datei gespeichert werden.

Tabelle 8.6 Genaue Beschreibung der Block-Kommandos

Cursorbewegung, Kommando	Funktion
Block markieren Shift → ← ↓ ↑	Markiert einen Block durch Hervorheben, wenn der Cursor bewegt wird. Markierter Text wird in einer anderen Intensität angezeigt.
Wort als Block markieren Ctrl K T	Markiert ein einzelnes Wort als Block. Steht der Cursor innerhalb des Wortes, wird das Wort markiert. Ist der Cursor außerhalb des Wortes, wird das Wort links vom Cursor markiert.
Block kopieren Ctrl Ins oder *Edit/Copy*, Shift Ins oder *Edit/Paste*	Kopiert einen vorher markierten Block in die Zwischenablage und fügt ihn an der aktuellen Cursorposition wieder ein. Beim Original-Block wird die Markierung entfernt, der Block dagegen, der in den Text eingefügt wurde, erscheint markiert. Ist kein Block markiert oder befindet sich der Cursor innerhalb des markierten Blocks, passiert nichts.
Block verschieben *Edit/Cut* oder Shift Del *Edit/Paste* oder Shift Ins	Fügt einen vorher markierten Block in die Zwischenablage und dann an der Cursorposition ein. Der Block verschwindet an der ursprünglichen Position, die Markierungen bleiben an der neuen Position erhalten. Ist kein Block markiert, passiert nichts.
Block löschen *Edit/Clear*, Ctrl Del , Ctrl K Y	Löscht einen vorher markierten Block. Der gelöschte Block kann nicht wiederhergestellt werden. Verwenden Sie daher dieses Kommando mit Vorsicht.
Block speichern Ctrl K W	Schreibt einen vorher markierten Block in eine Datei. Der Block bleibt unverändert stehen, die Markierungen bleiben erhalten. Bei Wahl dieses Kommandos erscheint eine Abfrage nach dem Namen der Datei, in die geschrieben werden soll. Jeder gültige Dateiname ist zulässig. (Die voreingestellte Extension ist .PAS.) Bevorzugen Sie einen Dateinamen ohne Extension, hängen Sie an den Namen einen Punkt an.
Block einlesen Ctrl K R	Liest eine Datei oder einen Block an der Cursorposition in den aktuellen Text ein. Der eingelesene Text ist dann als Block markiert. Nach Wahl dieses Kommandos werden Sie nach dem Namen der Datei gefragt, die gelesen werden soll. Sie können mit Jokerzeichen eine Datei suchen. Die angegebene Datei kann jeden gültigen Dateinamen haben.
Blockmarkierung entfernen/ anzeigen Ctrl K H	Schaltet die sichtbare Markierung eines Blockes aus und an. Die Kommandos, die sich auf den Block beziehen (Kopieren, Verschieben, Löschen, Drucken, Einlesen und Speichern) funktionieren nur, wenn die Blockmarkierung sichtbar ist. Die Cursorbewegungen, die sich auf den Block beziehen (Sprung zum Anfang/ Ende des Blocks) werden immer durchgeführt.

Cursorbewegung, Kommando	Funktion
Block drucken Ctrl K P	Druckt den im aktiven Editorfenster markierten Block.
Drucken *File/Print*	Druckt die Datei im aktiven Editorfenster.

Weitere Editor-Kommandos

Die nächste Tabelle beschreibt einige Editor-Kommandos genauer.

Tabelle 8.7 Weitere Editor-Kommandos

Cursorbewegung, Kommando	Funktion
Compiler-Parameter Ctrl O O	fügt die momentan gesetzten Parameter für den Compiler (Schalter, Befehle, Speichergrößen usw.) am Anfang des Quelltextes im gerade aktiven Editorfenster ein.
Automatische Einrückung *Options/ Environment/ Editor*	Öffnet das Fenster *Editor Options*, das das Markierungsfeld *Autoindent mode* enthält. Ist *Autoindent* gewählt, wird die Einrückung der aktuellen Zeile bei jeder folgenden Zeile wiederholt, d.h. wenn Sie Enter eingeben, springt der Cursor nicht in Spalte 1, sondern zur Start-Spalte der vorhergehenden nicht leeren Zeile. Wählen Sie zum Verändern der Einrückung die neue Spalte mit [Spacebar] oder ↑, die Voreinstellung ist *On*.
Marke suchen Ctrl Q n	Findet bis zu 10 Platzmarken (n steht für eine Zahl zwischen 0 und 9). Zu einer Marke gelangen Sie, indem Sie Ctrl Q drücken und die Nummer der Marke eingeben.
Neue Datei *File/New*	Öffnet ein neues Fenster
Datei öffnen F3, *File/Open*	Lädt eine bestehende Datei in ein Editorfenster.
Editor verlassen Alt X, *File/Exit*	Verläßt Turbo Pascal. Dabei erscheint eine Abfrage, ob Sie speichern möchten.
Zeile wiederherstellen *Edit/Restore line*	Macht die Änderungen in der zuletzt bearbeiteten Zeile rückgängig. Die Zeile wird in den Zustand zurückversetzt, in dem Sie vor der Bearbeitung war, gleichgültig, welche Änderung Sie vorgenommen haben. Das Kommando bezieht sich nur auf die zuletzt geänderte oder gelöschte Zeile.

Cursorbewegung, Kommando	Funktion
Datei sichern F2, *File/Save*	Sichert die Datei und kehrt in den Editor zurück.
Marke setzen Ctrl K n	Sie können bis zu 10 Stellen im Text markieren, indem Sie Ctrl K drücken und danach eine Nummer (0-9) eingeben. Ist eine Stelle markiert, gelangen Sie mit Ctrl Q n von jedem Ort der Datei an die markierte Stelle zurück. In jedem Fenster können Sie 10 Markierungen setzen.
Tabulator Tab	Setzt den Cursor auf die nächste Tabulatorposition. Voreingestellt ist ein Tab-Abstand von acht Zeichen.
Tabulator-Modus *Options/ Environment/ Editor*	Öffnet das Dialogfenster *Editor Options*, das das Markierungsfeld *Use tab character* enthält. Ist die Option gewählt, können Sie Tab-Zeichen eingeben (ASCII-Zeichen 8), ist sie nicht gewählt, werden automatisch ebenso viele Leerstellen eingefügt.

Suchen und Ersetzen

Mit den Kommandos *Search/Find* und *Search/Replace* können Sie Zeichenfolgen suchen und ersetzen lassen.

Der Suchbegriff kann dabei aus beliebigen Zeichen, einschließlich Kontrollzeichen, bestehen. Kontrollzeichen leiten Sie mit Ctrl P ein. Beispiel: Wenn Sie Ctrl T eingeben wollen, so drücken Sie Ctrl P, halten die Ctrl-Taste gedrückt und geben T ein. Sie fügen einen Zeilenumbruch in einem Such-Begriff ein, indem Sie Ctrl M (Carriage Return) eingeben.

Im folgenden werden die Arbeitsschritte zur Ausführung einer Suchen- und Ersetzen-Operation beschrieben:

Suchen und die Suche wiederholen

1. Wählen Sie *Search/Find*. Damit öffnen Sie das Dialogfenster *Find*.
2. Geben Sie in das Eingabefeld *Find* den Suchbegriff ein.
3. Wenn nötig, können folgende Such-Optionen angegeben werden:

∩ Mit dem Schalter *Direction* wählen Sie die Suchrichtung (vorwärts/rückwarts);
∩ mit dem Schalter *Scope* wählen Sie den Bereich, in dem gesucht werden soll;
∩ mit dem Schalter *Origin* wählen Sie den Ausgangspunkt der Suche (Cursorposition oder Dateianfang);
∩ im Markierungsfeld *Options* können Sie einstellen, ob nur ganze Wörter und/oder auch reguläre Ausdrücke gesucht werden sollen und ob dabei Klein- und Großschreibung berücksichtigt werden soll.

Die einzelnen Optionen wählen Sie mit Tab oder mit der Maus. Mit ← und

↵ setzen Sie die Schalter, und mit Space ändern Sie die Einträge der Markierungsfelder.

4. Starten Sie die Suche mit *OK* oder verwerfen Sie die Eingaben mit der Option *Cancel*.

5. Wollen Sie denselben Begriff nochmals suchen, so verwenden Sie *Search/Search again*.

Suchen und Ersetzen

1. Öffnen Sie das *Replace*-Dialogfenster durch Wahl von *Search/Replace*.

2. Geben Sie in das Eingabefeld *Find* den Suchbegriff ein.

3. Wählen Sie mit Tab oder mit der Maus das Eingabefeld *New* und geben Sie den Begriff ein, der den Suchbegriff ersetzen soll.

4. Setzen Sie die Optionen, und gehen Sie dabei wie im *Find*-Dialogfenster vor.

5. Starten Sie die Suche mit *OK* oder mit *Change all* oder verwerfen Sie die Eingaben mit *Cancel*. *Change all* ersetzt den Suchbegriff im gesamten voreingestellten Bereich.

6. Esc stoppt die Operation.

Begrenzerpaare suchen

Wenn Sie sich Ihre Quelldateien ansehen, stellen Sie fest, daß diese viele Ausdrücke in Klammern, verschachtelte Kommentare und andere Konstrukte mit Begrenzerzeichen enthalten, zum Beispiel:

- geschweifte Klammern: { }
- runde Klammern: ( )
- eckige Klammern: [ ]
- Anführungszeichen: "
- Apostrophe: '

Es ist oft schwierig, das zu einem Begrenzerpaar gehörige zweite Zeichen zu finden. Nehmen wir an, Sie haben ein kompliziertes Listing mit ineinander verschachtelten Ausdrücken und wollen überprüfen, ob alle runden Klammern richtig gesetzt wurden.

Mit den im Editor von Turbo Pascal verfügbaren Befehlen zum Suchen von Begrenzerzeichen ist dies ein Kinderspiel. Folgende Arbeitsschritte sind auszuführen:

1. Setzen Sie den Cursor auf das Begrenzerzeichen, zu dem der Partner gesucht werden soll (zum Beispiel auf die offene geschweifte Klammer eines mehrzeiligen Kommmentars).

2. Lassen Sie nun das zugehörige Begrenzerzeichen suchen, indem Sie Ctrl Q − drücken. (In unserem Beispiel sollte sich der Partner am Ende der Funktionsdefinition befinden.)

3. Der Editor positioniert nun den Cursor auf das nächste entsprechende Begrenzerzeichen. Mit dieser Vorgehensweise können Sie sicherstellen, daß der eingegebene Code keine alleinstehenden Begrenzer enthält. Zeigt er das zweite Begrenzerzeichen an der falschen Stelle an, wissen Sie, daß ein Begrenzerzeichen fehlt. Suchen Sie dann die Fehlerquelle.

Damit kennen Sie nun die grundlegenden Kommandos in Turbo Pascal zum Suchen von Begrenzerzeichen, es fehlen jedoch noch einige Details, die Ihnen die Arbeit zusätzlich erleichtern können:

- Es gibt zwei verschiedene Editor-Kommandos zum Suchen der Begrenzerzeichen: mit (Ctrl Q − können Sie vorwärts suchen lassen und mit Ctrl Q + rückwärts.
- Gibt es kein zugehöriges Begrenzerzeichen, wird der Cursor nicht bewegt.

Suche mit und ohne Richtung

Da nicht alle Begrenzerzeichen eindeutig erkennen lassen, in welcher Richtung das zugehörige Begrenzerzeichen liegt, gibt es für die Suche zwei Kommandos.

Nehmen wir an, Sie suchen das zugehörige Endezeichen für { oder für [. Hier kann der Editor erkennen, daß der Partner weiter unten stehen muß, d.h. er sucht vorwärts. Entsprechend dazu sucht er die zugehörigen Zeichen für) oder } rückwärts.

Bei Apostrophen und Anführungszeichen kann der Editor die erforderliche Suchrichtung jedoch nicht erkennen, hier müssen Sie diese also explizit angeben. Mit dem Kommando Ctrl Q − können Sie vorwärts suchen lassen, mit Ctrl Q + rückwärts.

In der nachstehenden Tabelle finden Sie alle möglichen Begrenzerpaare mit der Kennzeichnung, ob die Suchrichtung automatisch festgelegt ist und ob sie einen eindeutigen Partner haben.

Tabelle 8.8 Begrenzerpaare

Begrenzerpaare		Suchrichtung erkennbar?	verschachtelbar?
{	}	Ja	Ja
(	)	Ja	Ja
[	]	Ja	Ja
"	"	Nein	Nein
'	`	Nein	Nein

Verschachtelbare Begrenzer

Verschachtelbar bedeutet: Wenn der Editor den Partner zu einem Begrenzerzeichen sucht, bei dem die Suchrichtung festgelegt ist, so berücksichtigt er untergeordnete Paare innerhalb der Suche. Dies läßt sich am besten an Beispielen verdeutlichen:

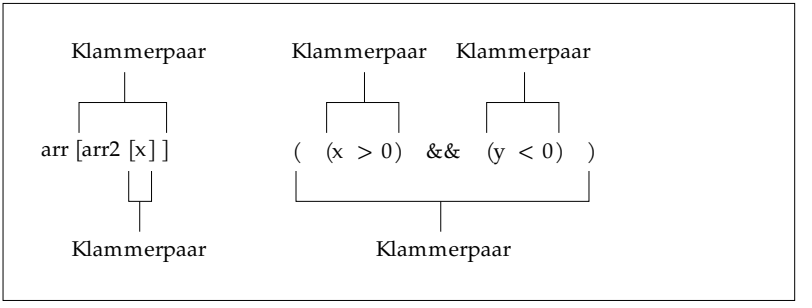


Abbildung 8.1 Suche nach den zugehörigen Begrenzerzeichen für [*und* (

9. Die Kommandozeilen-Version des Compilers

Obwohl auch die älteren Versionen von Turbo Pascal mit einer Menüstruktur arbeiten, soll es *immer noch* Leute geben, die mit einem eigenen (möglichst zeilenorientierten) Editor arbeiten, auf Batch-Dateien schwören und integrierte Entwicklungsumgebungen für etwas ähnlich Dekadentes wie eine gewisse Sorte von Nagetieren halten.

Damit auch diese Traditionalisten zufrieden sind, kommt auch die Version 6.0 des Turbo Pascal-Compilers in zwei Gewändern: Der integrierten Entwicklungsumgebung (TURBO.EXE) und der Kommandozeilen-Version (TPC.EXE). So verschieden die Arbeit mit beiden Versionen naturgemäß auch ist – intern sind sie hundertprozentig identisch.

Hilfe? Hilfe!

Ein Bonbon haben wir allerdings anzubieten, von dem wir mit Fug und Recht annehmen, daß es auch für wirkliche Puristen eine echte Erleuchtung ist: Nach dem Start des Zusatzprogramms THELP steht Ihnen die Syntaxhilfe der integrierten Entwicklungsumgebung auch bei der Arbeit mit TPC zur Verfügung. (Wie das funktioniert? THELP ist ein speicherresidentes Programm, das sich nach seiner Installation mit einem Druck auf die Taste 5 des numerischen Ziffernblocks aktivieren läßt und mit jedem ASCII-Editor zusammenarbeiten kann – unabhängig davon, ob es sich bei diesem Editor nun um EDLIN, HyperStar, WordStar oder um etwas Selbstgeschriebenes handelt).

Zu jeder Bibliotheksfunktion (wie *Writeln* oder *InitGraph*), jedem reservierten Wort (wie **uses** oder **while**) und jeder Unit (wie *Graph* oder *System*) existiert eine kurze Beschreibung. Natürlich ist auch diese Hilfestellung kontextbezogen – welche Hilfe Sie erhalten, hängt davon ab, wo im Quelltext eines Programms sich der Cursor befindet.

Probieren Sie es einmal aus: Setzen Sie das Verzeichnis als Standard, in dem TURBO.EXE gespeichert ist, und geben Sie den folgenden Befehl ein:

```
THELP
```

Nachdem Sie die Installationsmeldung von THELP erhalten haben, starten Sie Ihren Editor (wobei Sie gegebenfalls zuvor das Verzeichnis wechseln können), tippen

```
Writeln('Hallo');
```

und setzen Sie den Cursor danach so, daß er sich innerhalb des Wortes `Writeln` befindet. Ein Druck auf die Taste 5 bringt eine Beschreibung dieser Bibliotheksroutine auf den Bildschirm, wobei der Text wie bei der Hilfestellung zu den Menüpunkten der integrierten Entwicklungsumgebung organisiert ist: mit den Cursortasten können Sie zwischen den hervorgehoben dargestellten Punkten wählen; ein Druck auf Enter bringt Sie zum gewählten Punkt. Das Umblättern zwischen mehreren Textseiten geschieht mit PgUp und PgDn, Esc bringt Sie zu Ihrem Editor zurück.

Hinweis: Wenn sich der Cursor auf keinem bekannten Pascal-Schlüsselwort befindet, erhalten Sie das Inhaltsverzeichnis von THELP, über das Sie zwischen allen vorgegebenen Oberpunkten wählen können.

Weitere Details zu THELP finden Sie in Kapitel 10 »Die Zusatzprogramme«.

Der Aufruf von TPC.EXE...

geschieht von der DOS-Kommandoebene aus und hat die folgende Syntax:

```
tpc [Parameter/Schalter] Dateien
```

Dateien sind die Namen der zu compilierenden Dateien. Falls *Dateien* keinen Punkt (.) enthält, hängt TPC automatisch die Extension .PAS an: *TEST* wird also zu TEST.PAS, *TEST.XXX* bleibt unverändert, *TEST.* ebenfalls.

Jeder *Parameter* für TPC besteht aus einem Schrägstrich (/), gefolgt von einem Buchstaben und eventuell weiteren Angaben (wie z.B. einem Suchweg). Ein *Schalter* für TPC besteht ebenfalls aus einem Schrägstrich, darauf folgt ein Dollarzeichen ($), ein Buchstabe für den Namen des Befehls und schließlich ein Plus- oder Minuszeichen.

Parameter und Schalter können in beliebiger Folge vor und nach dem Namen der zu compilierenden Datei erscheinen; der Aufruf von TPC ohne jeden Parameter (also `tpc`Enter) gibt eine Liste aller möglichen Parameter zusammen mit einer Kurzbeschreibung aus.

Wenn es sich bei dem zu compilierenden Quelltext um eine Unit handelt, dann erzeugt TPC eine Datei mit dem als *Dateien* angegebenen Namen, aber der Extension .TPU. Ein Aufruf wie

```
tpc myunit
```

sucht das momentan gesetzte Verzeichnis nach dem Quelltext MYUNIT.PAS ab und erzeugt die Datei MYUNIT.TPU (vorausgesetzt, die Compilierung verläuft fehlerfrei).

Wenn es sich bei dem zu compilierenden Quelltext um ein Programm handelt, dann erzeugt TPC eine Datei mit dem als *Dateien* angegebenen Namen, aber der Extension .EXE – hier wird (im Gegensatz zur Compilierung von Units) über den Linker ein lauffähiges Programm erzeugt. Apropos: Der in TPC eingebaute Linker ist genauso »intelligent« wie sein Gegenstück in der integrierten

Entwicklungsumgebung und beseitigt nicht nur »toten« Code, sondern auch unbenutzte Datendeklarationen automatisch (vgl. Kapitel 14 »Interne Details« des Programmierhandbuches).

Parameter und Schalter

Die meisten Parameter des Compilers, die in der integrierten Entwicklungsumgebung über Menüpunkte gesetzt werden, lassen sich auch zusammen mit der Kommandozeilen-Version verwenden. Tabelle 9.1 gibt die Zusammenhänge wieder:

Tabelle 9.1 Kommandozeilen-Parameter vs. Menüpunkte

Parameter	Menüpunkt von TURBO.EXE	Einstellung
`/$A+`	*O/C/Word Align data*	Word
`/$A-`	*O/C/Word Align data*	Byte
`/$B+`	*O/C/Complete Boolean evaluation*	Complete
`/$B-`	*O/C/Complete Boolean evaluation*	Short circuit
`/$D+`	*O/C/Debug information*	On
`/$D-`	*O/C/Debug information*	Off
`/$E+`	*O/C/Emulation*	On
`/$E-`	*O/C/Emulation*	Off
`/$F+`	*O/C/Force far calls*	On
`/$F-`	*O/C/Force far calls*	Off
`/$G+`	*O/C/286 Instructions*	On
`/$G-`	*O/C/286 Instructions*	Off
`/$I+`	*O/C/I/O checking*	On
`/$I-`	*O/C/I/O checking*	Off
`/$L+`	*O/C/Local symbols*	On
`/$L-`	*O/C/Local symbols*	Off
`/$Msss,min,max`	*O/Memory sizes*	
`/$N+`	*O/C/Numeric processing*	8087/80287
`/$N-`	*O/C/Numeric processing*	
`/$O+`	*O/C/Overlays allowed*	On
`/$O-`	*O/C/Overlays allowed*	Off
`/$R+`	*O/C/Range checking*	On
`/$R-`	*O/C/Range checking*	Off
`/$S+`	*O/C/Stack checking*	On
`/$S-`	*O/C/Stack checking*	Off
`/$V+`	*O/C/Strict Var-string*	On
`/$V-`	*O/C/Strict Var-string*	Off
`/$X+`	*O/C/Extended Syntax*	On
`/$X-`	*O/C/Extended Syntax*	Off
`/B`	*Compile/Build*	

Parameter	Menüpunkt von TURBO.EXE	Einstellung
`/D`*Definition*	*O/C/Conditional defines*	
`/E`*Pfad*	*Options/Directories/EXE & TPU directory*	
`/F`*Seg:Ofs*	*Search/Find Error*	
`/GD`	*Options/Linker/Map file*	Detailed
`/GP`	*Options/Linker/Map file*	Publics
`/GS`	*Options/Linker/Map file*	Segments
`/I`*Suchweg*	*Options/Directories/Include directories*	
`/L`	*Options/Linker/Link buffer*	Disk
`/M`	*Compile/Make*	
`/O`*Suchweg*	*Options/Directories/Object directories*	
`/Q`	»Quiet« – keine Ensprechung	
`/T`*Suchweg*	*Options/Directories/Unit directories*	
`/U`*Suchweg*	*Options/Directories/Unit directories*	
`/V`	*Debugger/Standalone*	On

Hinweis: Zwischen Groß- und Kleinschreibung wird nicht unterschieden.

Schalter und Parameter für TPC werden entweder mit einem Schrägstrich (/) oder einem Bindestrich eingeleitet. Da Bindestriche (im Gegensatz zu Schrägstrichen) auch in Dateinamen zulässig sind, erwartet TPC hier jeweils ein vorangestelltes Leerzeichen, um sicher zwischen Dateinamen und Parametern unterscheiden zu können. Die folgenden Kommandozeilen sind also identisch:

```
tpc -ic:\tp\tvision -xnames.dta sortname -$r- -$f+
tpc /ic:\tp\tvision/xnames.dta sortname /$r-/$f+
```

(Natürlich könnte auch der zweite Aufruf trennende Leerzeichen enthalten, erforderlich sind sie aber nicht).

Compiler-Befehle

Beide Versionen des Compilers unterstützen Compiler-Befehle der Formate

```
{$<Befehlsname>+}
{$<Befehlsname>-}
{$<Befehlsname> Parameter}
```

innerhalb des Quelltextes (wobei *<Befehlsname>* für einen einzelnen Buchstaben steht und *Parameter* für einen Dateinamen oder Suchweg). Innerhalb der integrierten Entwicklungsumgebung können derartige Befehle zusätzlich über die Wahlpunkte des Menüs *Options/Compiler* gegeben werden; die Entsprechungen der Kommandozeilen-Version finden Sie in Tabelle 9.1. Ein Aufruf von TPC wie

```
tpc testprog /$r-
```

compiliert die Datei TESTPROG.PAS folglich mit ausgeschalteter Bereichsprüfung (*Range checking...Off*); beim Aufruf

```
tpc testprog /$r+
```

erzeugt der Compiler dagegen zusätzlichen Code zur Überprüfung von Wertebereichen. Natürlich kann eine Kommandozeile auch mehrere Compiler-Befehle enthalten:

```
tpc testprog /$r-/$i-/$v-/$f+
```

Die Trennung mehrerer Compiler-Befehle durch Kommas ist in derselben Weise wie innerhalb eines Quelltextes erlaubt – auch in der Kommandozeile ist hier nur eine einzige Einleitung notwendig:

```
tpc testprog /$r-,i-,v-,f+
tpc testprog -$r-,i-,v-,f+
```

Die Angabe der Bereichsgrößen für Stack und Heap stellt die einzige Ausnahme dar, weil dieser Compiler-Befehl selbst Kommas enthält:

```
/$m Stack,HeapMin,HeapMax
```

Stack steht hier für die Größe des Stack, *HeapMin* für die minimale und *HeapMax* für die maximale Größe des Heap. Alle drei Angaben haben die Einheit *Byte* und können entweder in dezimaler oder in hexadezimaler Form erfolgen:

```
tpc testprog /$m16384,0,655360
tpc testprog /$m$4000,0,$A0000
tpc testprog /$r-,i- /$m$4000,0,30000 /$v-,f+
```

Der bedingte Compiler-Befehl DEFINE

Über die Option `/D`lassen sich bedingte Symbole definieren, die dem Compiler-Befehl ($DEFINE *Symbolname*) bzw. der Menü-Option *Options/Compile/Conditional defines* in der IDE entsprechen. Der Angabe von /D folgen ein oder mehrere bedingte Symbole, die durch Semikolons voneinander getrennt sind:

```
tpc testprog /diocheck;debug;list
```

Zulässig ist auch folgende Variante:

```
tpc testprog /diocheck/ddebug/dlist
```

In diesen beiden Kommandozeile werden für die Compilierung von TESTPROG.PAS drei Symbole (*iocheck*, *debug* und *list*) definiert. Dies ist gleichbedeutend mit folgenden Definitionen am Anfang von TESTPROG.PAS.

```
{$DEFINE IOCHECK}
{$DEFINE DEBUG}
{$DEFINE LIST}
```

Die Art der Compilierung

Über die drei Zusätze /M (*Make*), /B (*Build*) und /F (*Find error*) läßt sich der globale Ablauf der Compilierung festlegen, mit /Q (*Quiet*) werden Meldungen über Zeilennummern unterdrückt. (Wie bei allen anderen Parametern können Sie auch hier mit Bindestrichen anstelle von Schrägstrichen arbeiten, dürfen dann aber die trennenden Leerzeichen nicht vergessen).

Der Zusatz »Make« (/M)
aktiviert die in TPC eingebaute MAKE-Funktion, die wie in der integrierten Entwicklungsumgebung arbeitet: TPC prüft vor jeder Aufnahme einer Unit den dazugehörigen Quelltext und recompiliert zuerst einmal die Unit, falls deren Quelltext neueren Datums sein sollte als ihr Objectcode. Bei der Recompilierung einer Unit werden die Units geprüft, die diese Unit voraussetzt, und gegebenenfalls ebenfalls erneut übersetzt – mit dem Effekt, daß sämtliche an einem Programm beteiligten Units garantiert immer auf dem neuesten Stand der Entwicklung sind.

Der folgende Aufruf zeigt die Verwendung von /M:

```
tpc testprog /m
```

Hinweis: Units innerhalb von TURBO.TPL sind vom Zeit- und Datumsvergleich ausgeschlossen – TPC sucht also nicht nach SYSTEM.PAS, OVERLAY.PAS usw.

Der Zusatz »Build all« (/B)
Jede an der Compilierung beteiligte Datei wird ohne vorherige Prüfung von Datum /Uhrzeit neu compiliert (soweit ein entsprechender Quelltext vorhanden ist. Sinnvoll ist diese Option beispielsweise dann, wenn Sie mit bedingter Compilierung arbeiten und nur einige Symbole (nicht aber Teile der Quelltexte) verändert haben.

Der folgende Aufruf von TPC zeigt die Verwendung von /B:

```
tpc testprog /b
```

Hinweis: /M und /B schließen sich gegenseitig aus. Falls Sie versehentlich doch beide Parameter angeben, benutzt TPC die jeweils letzte (rechteste) Angabe der Kommandozeile.

Der Zusatz »Find error« (/F)
Wenn ein Programm durch einen Laufzeitfehler abgebrochen wird, erhalten Sie einen Fehlercode und die Adresse des Befehls, der den Fehler ausgelöst hat. Die Recompilierung des Programms mit dem Zusatz /F und der Angabe dieser Adresse lokalisiert die entsprechende Stelle im Quelltext – vorausgesetzt, daß die Informationen für den Debugger nicht mit dem Compiler-Befehl {$D-} im Quelltext unterdrückt wurden.

Zur Demonstration erzeugen wir eine Datei namens TEST.PAS, die das folgende Programm enthält:

```
program Oha;
var
  i : Integer;
begin
  i := 0;
  i := i div i;  { das ist eine Division durch Null! }
end.
```

Nach der Compilierung dieses Programms als TEXT.EXE mit

```
tpc test
```

sollte der Befehl *dir test.** mindestens zwei Dateien zeigen: den Quelltext TEST.PAS und das ausführbare Programm TEST.EXE. Starten wir die letztere der beiden:

```
c:\>test
Runtime error 200 at 0000:0011
```

Der von uns absichtlich erzeugte Fehler hat also den Code 200 (siehe Kapitel 3 »Fehlermeldungen und -codes« des *Referenzhandbuches*) und wurde durch den Befehl ausgelöst, der sich im Segment 0000 des Programms auf der (hexadezimalen) Adresse 0010 befindet.

Da wir die Informationen für den Debugger nicht explizit unterdrückt haben, verläuft die Zuordnung der entsprechenden Stelle im Quelltext recht einfach – sie besteht aus dem Aufruf von TPC mit dem Zusatz /F und der Angabe der Fehleradresse.

```
c:\>tpc test /f0:11
Turbo Pascal Version 6.0
Copyright (C) Borland International
TEST.PAS(7)
TEST.PAS(6): Target address found.
  i := i div i;  { das ist eine Division durch Null! }
  ^
```

Das Caret steht immer am Anfang des »fehlerhaften« Befehls und ermöglicht eine eindeutige Zuordnung auch dann, wenn der Quelltext mehrere Befehle in einer Zeile enthält.

Hinweis: Wenn Sie anstelle einer »Ziel-Adresse« die Meldu*ng Error 117: Target address not found* erhalten, sollten Sie prüfen, ob

- sich der Fehler innerhalb einer mit {$L} eingebundenen Maschinenroutine befindet (die keine Informationen über »Zeilennummern« enthält);
- nicht im Quelltext des Programms oder etwa in der Konfigurationsdatei TURBO.CFG der Compiler-Befehl {$D-} bzw. der Kommandozeilen-Parameter /$D- enthalten ist;
- Sie exakt dieselben Parameter und Schalter angegeben haben wie bei der ursprünglichen Compilierung des Programms. Durch die Informationen zur Fehlersuche verändern sich die Adressen der einzelnen Befehle nicht – für an- und abgeschaltete Stack-, I/O- oder Bereichsprüfungen sieht die Sache dagegen

anders aus, weil der Compiler in diesen Fällen zusätzlichen Code erzeugt bzw. zuvor eingefügte Prüfroutinen hinauswirft.

»Quiet Mode« (/Q)
Normalerweise zählt TPC während der Compilierung die Zeilennummern des Quelltextes auf dem Bildschirm mit und mißt die für die Compilierung benötigte Zeit über die eingebaute Uhr des Computers. Ein Aufruf zusammen mit /Q unterdrückt sowohl die Zählung als auch die Zeitmessung: TPC gibt nur noch die Namen der beteiligten Dateien und die Gesamt-Zeilenzahl aus. Wenn ein Fehler auftritt, wird er gemeldet.

Verzeichnisse und Suchwege

TPC kennt die Wahlpunkte des Menüs *Options/Directories* als Kommandozeilen-Parameter: über entsprechende Angaben läßt sich bestimmen, wo erzeugte Dateien gespeichert werden sollen und wo der Compiler nach Konfigurationsdateien, Include-Dateien, Units und Object-Dateien sucht. (Auf den Punkt »Konfigurationsdateien« kommen wir später noch zurück).

»EXE & TPU Directory« (/E)
Dieser Parameter hat dieselbe Funktion wie der Wahlpunkt *O/D/EXE & TPU directory* der integrierten Entwicklungsumgebung. Er legt fest, in welchem Verzeichnis TPC erzeugte Dateien (.EXE und .TPU) speichert. Der Aufruf

```
tpc testprog /ec:\tp\exec
```

bestimmt, daß die Datei TESTPROG.EXE sowie im Verzeichnis TP\EXEC des Laufwerks C: gespeichert wird. Entsprechendes gilt für die Compilierung von Units.

»Include Directories« (/I)
Dieser Parameter hat dieselbe Funktion wie der Wahlpunkt *O/D/Include directories* der integrierten Entwicklungsumgebung. Er legt fest, in welchen Verzeichnissen TPC über den Compiler-Befehl {$I *Dateiname*} aufzunehmende Dateien sucht. Der Aufruf

```
tpc testprog /ic:\tp\include
```

bestimmt, daß in TESTPROG.PAS mit {$I} angegebene Quelltexte zuerst im momentan gesetzten Verzeichnis und danach im Verzeichnis TP\INCLUDE des Laufwerks C: gesucht werden.

Wie der entsprechende Wahlpunkt der integrierten Entwicklungsumgebung erlaubt auch der Parameter /I die Angabe mehrerer Suchwege, die durch Semikolons getrennt werden. Alternativ dazu läßt sich /I mehrfach in einer Kommandozeile angeben – die folgenden Aufrufe von TPC sind also gleich:

```
tpc testprog /ic:\tp\include;c:\tp\biblio
tpc testprog /ic:\tp\include/ic:\tp\biblio
```

»Object Files Directories« (/O)

Dieser Parameter hat dieselbe Funktion wie der Wahlpunkt *O/D/Object directories* der integrierten Entwicklungsumgebung. Er legt fest, in welchen Verzeichnissen TPC Dateien sucht, die über den Compiler-Befehl {$L *Dateiname*} aufgenommen werden sollen. Der Aufruf

```
tpc testprog /oc:\tp\asm
```

bestimmt, daß in TESTPROG.PAS mit {$L} angegebene Object-Dateien zuerst im momentan gesetzten Verzeichnis und danach im Verzeichnis TP\ASM des Laufwerks C: gesucht werden.

Wie der entsprechende Wahlpunkt der integrierten Entwicklungsumgebung erlaubt auch der Parameter /O die Angabe mehrerer Suchwege, die durch Semikoli voneinander getrennt werden. Alternativ dazu läßt sich /O mehrfach in einer Kommandozeile angeben – folgende Aufrufe von TPC sind also gleich:

```
tpc testprog /oc:\tp\asm;c:\tp\asmlib
tpc testprog /oc:\tp\asm/oc:\tp\asmlib
```

»Unit Directories« (/U)

Dieser Parameter hat dieselbe Funktion wie der Wahlpunkt *O/D/Unit directories* der integrierten Entwicklungsumgebung. Er legt fest, in welchen Verzeichnissen TPC Units sucht, die über die Anweisung **uses** aufgenommen werden sollen und nicht Teil von TURBO.TPU sind. Der Aufruf

```
tpc testprog /uc:\tp\units
```

bestimmt, daß in TESTPROG.PAS mit **uses** angegebene Units zuerst in TURBO.TPL, danach im momentan gesetzten Verzeichnis und schließlich im Verzeichnis TP\UNITS des Laufwerks C: gesucht werden.

Wie der entsprechende Wahlpunkt der integrierten Entwicklungsumgebung erlaubt auch der Parameter /U die Angabe mehrerer Suchwege, die durch Semikolons voneinander getrennt werden. Alternativ dazu läßt sich /U mehrfach in einer Kommandozeile angeben – folgende Aufrufe von TPC sind also gleich:

```
tpc testprog /uc:\tp\units;c:\tp\screenio
tpc testprog /uc:\tp\units/uc:\tp\screenio
```

/V – Der Schalter für den Turbo Debugger

Beim Aufruf mit dem Schalter /V erzeugt TPC.EXE Informationen zur Fehlersuche mit dem *Turbo Debugger*, die (analog zu den Informationen für den integrierten Debugger) als Teil der .EXE- bzw. .TPU-Datei(en) gespeichert werden. Auch hier werden die Dateien entsprechend vergrößert – auf die Ausführungsgeschwindigkeit der Programme haben die zusätzlichen Informationen zur Fehlersuche aber nicht den geringsten Einfluß.

Parameter für den Linker

/GP, /GS und /GD - Erzeugen einer .MAP-Datei

Diese Schalter (sie haben ihre Entsprechung in dem Menüpunkt *O/L/Map File*) legen fest, ob bei der Compilierung eines Programms eine zusätzliche .MAP-Datei erzeugt wird und wie umfangreich die in dieser Datei gespeicherten Informationen sind.

Tabelle 9.2 faßt die Wirkungen und die Äquivalente der integrierten Entwicklungsumgebung von /GS, /GP und /GD zusammen:

Tabelle 9.2 : Kommandozeilen-Parameter für den Linker

Parameter	Beschreibung
(keine Angabe)	entspricht dem Menüpunkt *O/L/Map file/none* der integrierten Entwicklungsumgebung – es wird keine .MAP-Datei erzeugt
/GS	»Segments« – TPC.EXE erzeugt eine .MAP-Datei, die denselben Namen wie die .EXE-Datei bekommt und nur die Namen, Größen und Adressen der Segmente des Programms auflistet
/GP	»Publics« – TPC.EXE erzeugt eine .MAP-Datei mit Namen und Adressen aller globalen Bezeichner – vorausgesetzt, daß *Debug information* auf der Standardvorgabe /$D+belassen wird
/GD	»Detailed« – TPC.EXE erzeugt eine .MAP-Datei mit Namen und Adressen sämtlicher Bezeichner des Programms – vorausgesetzt, daß sowohl *Debug information* als auch *Local symbols* mit den Standardvorgaben arbeiten (/$D+ bzw. /$L+).

Hinweis: Äquivalente Compiler-Befehle im Quelltext für diese drei Schalter sind nicht definiert.

/L – Link-Puffer

Beim Binden mehrerer Module zu einem lauffähigen Programm werden einige temporäre Tabellen und andere Daten erzeugt. /L legt fest, ob Turbo Pascal den Hauptspeicher oder eine separate Diskettendatei zur Zwischenspeicherung verwendet. Der Linker von Turbo Pascal arbeitet in zwei Schritten: zuerst werden sämtliche Routinen beteiligter Units (.TPU-Dateien) gekennzeichnet, die das Programm auch tatsächlich benutzt, der zweite Schritt bindet die gekennzeichneten Routinen und das Programm selbst zu einer .EXE-Datei.

Nur wenn der Schalter /L beim Aufruf von TPC.EXE angegeben ist, geschieht die Zwischenspeicherung der entsprechenden Informationen in einer temporären Datei auf der Diskette, die beim zweiten Arbeitsgang des Linkers wieder eingelesen wird. Dieses (langsamere) Verfahren spart einigen Platz im Hauptspeicher und ist nur bei sehr großen Programmen sinnvoll.

Im Normalfall (d.h. beim Aufruf ohne /L) wird der Hauptspeicher zur Zwischenspeicherung benutzt.

Die Datei TPC.CFG

TPC sucht bei jedem Start automatisch im momentan gesetzten Verzeichnis nach einer Datei namens TPC.CFG. Wenn die Suche erfolglos bleibt, wird beim Betrieb unter DOS 3.0 (oder einer höheren Version) zusätzlich das Verzeichnis abgesucht, in dem sich TPC.EXE selbst befindet. Mit dem Kommandozeilen-Parameter /T läßt sich für alle DOS-Versionen ein weiterer Suchweg für TPC.CFG festlegen – allerdings muß dieser Parameter dazu der allererste einer Kommandozeile sein.

TPC.CFG ist (im Gegensatz zur Konfigurationsdatei der integrierten Entwicklungsumgebung) eine völlig normale ASCII-Datei und kann mit jedem Editor bearbeitet werden.

Ein »typisches« Beispiel für eine Konfigurationsdatei von TPC:

```
/tc:\tp\bin
/uc:\tp\units
/$r-
```

Diese Parameter legen fest, daß die Unit-Bibliothek (TURBO.TPL) im Verzeichnis TP\BIN und einzelne Units (.TPU-Dateien) im Verzeichnis TP\UNITS des Laufwerks C: gesucht werden. Die Compilierung erfolgt mit ausgeschalteter Bereichsprüfung. Ein Aufruf wie

```
tpc testprog
```

liest den Inhalt von TPC.CFG und stellt ihn den restlichen Kommandozeilen-Parametern voran, ergibt also die Kommandozeile

```
tpc /tc:\tp\bin /uc:\tp\units /$r- testprog
```

Da TPC die Wiederholung eines Parameters nicht als Fehler anmerkt, sondern einfach den jeweils zuletzt angegebenen Wert benutzt, lassen sich Inhalte einer Konfigurationsdatei durch Kommandozeilen-Parameter außer Kraft setzen. Der Aufruf

```
tpc testprog /$r+
```

ergibt zusammen mit TPC.CFG die Kommandozeile

```
tpc /tc:\tp\bin /uc:\tp\units /$r- testprog /$r+
```

Hier wird also *mit* aktivierter Bereichsüberprüfung gearbeitet. Dasselbe gilt auch für den Namen der zu compilierenden Datei: TPC verwendet nur den jeweils zuletzt aufgeführten Parameter – Ihre Konfigurationsdatei könnte also einen »Standard-Dateinamen« enthalten:

```
testprog
/$r-,i-,d+,t+
```

Jeder Aufruf von TPC ohne einen Dateinamen compiliert in diesem Fall die Datei TESTPROG.PAS; ein Aufruf *mit* einem Dateinamen wie

```
tpc meier
```

wird als

```
tpc testprog /$r-,i-,d+,t+ meier
```

gelesen – der Compiler bearbeitet also die Quelltext-Datei MEIER.PAS. In TPC.CFG lassen sich alle Parameter zusammenfassen, mit denen TPC arbeiten kann. Ein etwas umfangreicheres Beispiel zum Schluß:

```
/tc:\tp\bin
/uc:\tp\units;c:\tp\screenio
/oc:\tp\asm;c:\tp\screenio
/$R-,S-,V+,I-,D+
/$M$A000,0,0 /GD
```

Ein Aufruf von TPC wie

```
tpc testprog
```

erzeugt eine Kommandozeile, die in diesem Handbuch gar nicht mehr auf eine einzelne Zeile paßt:

```
tpc /tc:\tp\bin /uc:\tp\units;c:\tp\screenio /oc:\tp\asm;c:\tp\
screenio /$R-,S-,V+,I-,D+ /$M$A000,0,0 /GD testprog
```

Hier wird also die Unit-Bibliothek (TURBO.TPL) im Verzeichnis TP\UNITS des Laufwerks C: gesucht, einzelne Units (.TPU-Dateien) in den Verzeichnissen TP\SCREENIO und TP\UNITS und Object-Dateien in den Verzeichnissen TP\SCREENIO und TP\ASM. Der Compiler arbeitet ohne Bereichs-, Stack- und I/O-Prüfung, erzeugt Informationen zur Fehlersuche mit dem integrierten Debugger, stellt $A000 Bytes für den Stack zur Verfügung und reserviert überhaupt keinen Platz für dynamische Variablen. Bei einer erfolgreichen Compilierung erzeugt der Linker schließlich eine Datei namens TESTPROG.MAP, in der sämtliche Symbole des Programms verzeichnet sind.

Compilieren im Protected Mode

Wenn Sie mit einem 286-, 386- oder 486-Rechner mit mindestens 1MByte Extended Memory arbeiten, können Sie TPCX.EXE benutzen. TPCX arbeitet im Protected Mode und greift auf das Extended Memory zu; es kann daher sehr große Programme compiliert (Um Mißverständnissen vorzubeugen: Die mit TPCX erzeugten Programme laufen im Real Mode, sind also *keine* Protected-Mode-Anwendungen. Beachten Sie, daß TPCX nicht mit EMS-Speicher arbeitet.

TPCX ist wesentlich größer als TPC, und Programme, die im Protected Mode arbeiten, benötigen wesentlich mehr Speicher als Programme im Real Mode. Benutzen Sie immer dann TPC, wenn Sie nicht auf die Speicherkapazitäten von TPCX zugreifen müssen. TPCX verfügt im übrigen über die gleichen Kommandozeilen-Optionen wie TPC.

10. Die Zusatzprogramme

Turbo Pascal wird zusammen mit einer ganzen Reihe weiterer Programme ausgeliefert, die dieses Kapitel in alphabetischer Reihenfolge erläutert.

BINOBJ: konvertiert (nicht nur) Grafiktreiber und Vektor-Zeichensätze in .OBJ-Dateien, d.h. ermöglicht ihre direkte Aufnahme in ein Turbo-Pascal-Programm

GREP: eine schnelle Version des gleichnamigen UNIX-Dienstprogramms zum Durchsuchen von Quelltexten

MAKE: »automatisiert« die Erstellung von Programmen, die aus mehreren Quelltexten bestehen

THELP: macht die Syntax-Hilfestellung der integrierten Entwicklungsumgebung in beliebigen ASCII-Editoren verfügbar

TOUCH: setzt Datum und Uhrzeit von Dateien direkt

TPUMOVER: ist die Bibliotheksverwaltung von Turbo Pascal – mit diesem Programm können Sie TURBO.TPL beliebig eigene Units hinzufügen (oder auch extrahieren, entfernen, zusammenfassen...)

BINOBJ – Konvertierung in .OBJ-Dateien

Mit BINOBJ lassen sich binäre Dateien in .OBJ-Dateien konvertieren, die ihrerseits über den Compiler-Befehl {$L} in ein Turbo-Pascal-Programm als »Prozedur« (d.h. ins Code-Segment) aufgenommen werden können. Abgesehen davon, daß sich so Daten (wie z.B. gespeicherte Bildschirme) aufnehmen lassen, deren Codierung als typisierte Konstante unnötig aufwendig wäre, liegt der Hauptzweck von BINOBJ in der Konvertierung von Grafik-Treibern und Zeichensätzen. (Tatsächlich haben .BGI- und .CHR-Dateien ein binäres Format). Ein Programm, das seine Treiber und Zeichensätze direkt enthält, kommt zur Laufzeit ohne weitere Dateien aus – wir haben das am Beispiel GRLINK.PAS demonstriert.

BINOBJ wird mit drei Parametern aufgerufen:

```
BINOBJ Quelldatei[.BIN] Zieldatei[.OBJ] Prozedurname
```

Quelldatei steht für den Namen der zu konvertierenden Datei, *Zieldatei* für die Datei, die BINOBJ erzeugen soll. *Prozedurname* steht für den Namen der »Prozedur«, unter dem die Zieldatei mit {$L} in das Programm aufgenommen wird, muß also ein gültiger Pascal-Bezeichner sein. Wenn die Angabe für *Quelldatei* kein Suffix enthält, fügt BINOBJ automatisch die Endung .BIN an; analoges gilt für die erzeugte Datei, die in diesem Fall automatisch das Suffix .OBJ erhält.

Das folgende Beispielprogramm liest eine Datei namens MENU.DTA und kopiert ihren Inhalt in den Bildspeicher (insgesamt 80 * 25 * 2 = 4000 Bytes). Damit das Programm so übersichtlich wie möglich bleibt, haben wir auf Fehlerprüfungen verzichtet:

```
program Traditionell;
uses Crt;
var
  MenuP : Pointer;
  MenuF : File;

procedure ShowScreen(var ScreenData : Pointer);
{ Kopiert 4000 Bytes von ScreenData^ in den Bildspeicher }
var  ScreenSegment : Word;

begin
  if (Lo(LastMode) = 7) then ScreenSegment := $B000  { Mono }
   else ScreenSegment := $B800;
  Move(ScreenData^,Ptr(ScreenSegment, 0)^, 4000);  { Kopieren }
end;

begin  { Hauptprogramm }
  Assign(MenuF, 'MENU.DTA');  Reset(MenuF, 1); { Öffnen der Datei }
  GetMem(MenuP, 4000);                 { Puffer auf dem Heap }
  BlockRead(MenuF, MenuP^, 4000);      { Lesen der Datei }
  Close(MenuF);
  ShowScreen(MenuP);                   { und Kopieren }
end.
```

Hier werden die Daten auf »traditionelle« Weise von der Diskette gelesen – MENU.DTA muß also zur Laufzeit des Programms verfügbar sein.

Alternativ dazu lassen sich die in MENU.DTA enthaltenen Daten mit den folgenden Schritten direkt in das Programm aufnehmen:

- Konvertierung von MENU.DTA in eine .OBJ-Datei:

```
BINOBJ menu.dta menudta MenuData
```

Dieser Aufruf von BINOBJ liest die Datei MENU.DTA, erzeugt die Datei MENUDTA.OBJ (das Suffix .OBJ wird automatisch angehängt) und gibt den Daten den »Prozedurnamen« *MenuData*.

- Danach wird das Programm um eine **external**-Deklaration und den entsprechenden Compiler-Befehl {$L} erweitert:

```
program Daten_direkt;
```

```
uses Crt;
    { <- Die Datei-Deklaration wird nicht mehr gebraucht }

procedure ShowScreen(var ScreenData : Pointer);  { unverändert }
{ Kopiert 4000 Bytes von ScreenData^ in den Bildspeicher }
var  ScreenSegment : Word;
begin
  if (Lo(LastMode) = 7) then ScreenSegment := $B000  { Mono }
   else ScreenSegment := $B800;
  Move(ScreenData^,Ptr(ScreenSegment, 0)^, 4000);    { Kopieren }
end;

 { "Prozedur" MenuData, als external deklariert und der Compiler-
   Befehl zur Aufnahme von MENUDTA.OBJ - in dieser Datei sind
   die Daten als "Prozedur MenuData" enthalten }
procedure MenuData; external;
{$L MENUDTA.OBJ}

begin   { Aufruf von ShowScreen mit der Adresse von MenuData }
  ShowScreen(@MenuData);
end.
```

Achtung: Hier wird die *Adresse* der »Prozedur« übergeben, d.h. die Adresse der als *MenuData* gespeicherten Daten.

Die Vorteile dieses Verfahrens liegen auf der Hand: Erstens kommt das Programm ohne zusätzliche Dateien aus, zweitens entfallen die entsprechenden Diskettenzugriffe – und drittens können wir die Daten über einen normalen Pascal-Bezeichner ansprechen.

Die Nachteile: Da die gespeicherten Daten Teil des Programms sind, wächst sein Umfang entsprechend – außerdem müssen Sie die Datei MENU.DTA nach jeder Veränderung erneut mit BINOBJ in eine .OBJ-Datei konvertieren.

BINOBJ ist also hauptsächlich dann sinnvoll, wenn die einzubindenden Daten »stabil« sind, d.h. aller Voraussicht nach nicht allzuoft verändert werden müssen. Das Programm GRLINK.PAS zeigt eine derartige Anwendung: Es bindet alle verfügbaren Grafiktreiber und Zeichensätze in ein Programm ein, das danach ohne zusätzliche .BGI- und .CHR-Dateien auf jedem Computer lauffähig ist, den *Graph* unterstützt.

Hinweis: Eine Beschreibung der Schritte zur Erstellung von GRLINK.EXE finden Sie in GRLINK.PAS, eine ausführliche Erläuterung der Funktionen *RegisterBGIFont* und *RegisterBGIDriver* in den entsprechenden Abschnitten von Kapitel 1 »Verzeichnis der Routinen« des *Referenzhandbuches*.

GREP – Absuchen von Quelltexten

GREP ist (wie BINOBJ) unabhängig von der integrierten Entwicklungsumgebung bzw. der Kommandozeilen-Version des Compilers: falls Sie beispielsweise bei einem riesengroßen Projekt (egal, in welcher Programmiersprache)

vergessen haben sollten, in welcher Ihrer 40 Quelltexte die Routine *ModemInitialize* definiert ist, können Sie das ansonsten fällige »Durchschaufeln« sämtlicher Dateien eines oder mehrerer Verzeichnisse beruhigt diesem Programm überlassen. Die Syntax des Aufrufs ist:

```
GREP [Schalter] Suchstring Dateiname(n)
```

Dateinamen

werden beim Aufruf von GREP als letztes Element der Kommandozeile angegeben und legen fest, welche Datei oder Dateigruppe(n) abgesucht werden soll. Ein oder mehrere Dateinamen können entweder direkt (TEST1.PAS TEST2.PAS), mit Jokerzeichen (TEST?.PAS oder T*.PAS), zusammen mit einem Suchweg und/oder einem Laufwerksbezeichner angegeben werden.

Wenn kein Suchweg angegeben ist, geht GREP vom momentan gesetzten Verzeichnis aus; falls Sie überhaupt keinen Dateinamen angeben, liest GREP die Standardeingabe des Systems (was ohne eine Umleitung der Ein-/Ausgabe von DOS nicht allzuviel Sinn macht).

Um unser Beispiel fortzusetzen: das Absuchen sämtlicher .PAS-Dateien im momentan gesetzten Verzeichnis geschieht durch den Befehl:

```
GREP -l ModemInitialize *.pas
```

Die Schalter von GREP

GREP verfügt über eine ganze Reihe von Schaltern, die (wie auch bei MAKE) mit einem (einzigen) Bindestrich eingeleitet werden. Darauf folgt der Name des Schalters (ein Buchstabe) und entweder ein Plus- oder ein Minuszeichen, mit dem der betreffende Schalter an- oder ausgeschaltet wird. Die Standardvorgabe für jeden *angegebenen* Schalter ist »+« – eine Angabe wie -r ist also äquivalent zu -r+. Die folgenden Schalter sind definiert:

-c »count« – für jede Datei, bei der zumindest eine Zeile mit *Suchstring* übereinstimmt, wird der Dateiname zusammen mit der Anzahl der übereinstimmenden Zeilen ausgegeben, weitere Ausgaben erfolgen nicht.

-d »directories« – Dateien werden nicht nur im momentan gesetzten bzw. angegebenen Verzeichnis gesucht, sondern auch in allen Unterverzeichnissen, die zu diesem Verzeichnis gehören.

-i »ignore uppercase« – GREP behandelt Groß- und Kleinbuchstaben gleich, wobei die deutschen Umlaute (wie üblich) nicht mit eingeschlossen sind – 'Ä' und 'ä' sind also auch in diesem Fall »verschiedene« Zeichen.

-n »number« – jede übereinstimmende Zeile wird zusammen mit einer Zeilennummer ausgegeben.

-l »list« – wenn *Suchstring* innerhalb einer Datei gefunden wird, gibt GREP nur den Namen dieser Datei aus und setzt die Suche sofort in der nächsten Datei fort (soweit *Dateiname* mehrere Dateien angibt).

-o »output for UNIX« – paßt die Ausgaben von GREP dem »pipelining« von UNIX an: jeder ausgegebenen Zeile wird der Name der entsprechenden Datei vorangestellt.

-r »regular expression« – der durch *Suchstring* angegebene Text wird nicht als direkte Angabe, sondern als regulärer Ausdruck behandelt (siehe nächster Abschnitt).

-u »update options« – GREP kombiniert die angegebenen Schalter mit den Standardvorgaben und *schreibt eine neue Version von sich selbst*, die mit diesen Parametern in Zukunft als Standardvorgabe arbeitet. Damit läßt sich das Programm jederzeit an eigene Bedürfnisse anpassen.

-v »vice versa« – nur *nicht übereinstimmende* Zeilen werden ausgegeben, d.h. Zeilen, die *Suchstring* nicht enthalten.

-w »word search« – mit dem Suchstring übereinstimmende Zeichenfolgen werden nur dann als »gefunden« betrachtet, wenn vor und nach ihnen ein Trennzeichen steht. Standardmäßig betrachtet GREP die Zeichen A..Z, a..z, 0..9 und den Unterstrich als Teil eines Wortes (d.h. nicht als Trennzeichen).

Mit einer Angabe der Form -w[Zeichensatz] läßt sich festlegen, welche Zeichen GREP als Teil eines Wortes betrachten soll (wobei Groß- und Kleinbuchstaben hier ohne Unterscheidung behandelt werden): mit -w[A-Z0-9_äöüÄÖÜß] ließe sich beispielsweise festlegen, daß GREP die deutschen Umlaute ebenfalls nicht als Trennzeichen interpretiert.

Hinweis: Angaben eines »Wort-Zeichensatzes« werden durch den Parameter -u ebenfalls gespeichert.

-z »verbose« – GREP gibt bei jeder durchsuchten Datei den Namen sowie die Anzahl der gefundenen Zeilen aus – auch wenn die Datei keine Übereinstimmungen enthält. Gefundene Zeilen werden mit einer vorangestellten Zeilennummer ausgegeben (wie -c -n).

Einige dieser Schalter arbeiten entgegengesetzt – deshalb gibt GREP eine Art »Rangfolge« vor:

```
-z -l -c -n
```

Wenn der Schalter -z angegeben ist, werden die Schalter l, c und n ignoriert, die Angabe -l setzt Vorgaben und Angaben von c und n außer Kraft usw.

Suchstrings und reguläre Ausdrücke

Normalerweise behandelt GREP den *Suchstring* als direkte Angabe, d.h. vergleicht ihn zeichenweise mit den jeweiligen Dateien. Falls *Suchstring* Leerzeichen und Tabs enthalten soll, müssen Sie die Angabe in Anführungszeichen

setzen: um beispielsweise das Wort »Kommentar « (mit einem folgenden Leerzeichen) zu suchen, lautet die Angabe

```
"Kommentar "
```

GREP behandelt Zeilenenden bei direkten Vergleichen grundsätzlich als Trennung. Die beiden Zeilen

```
Komm
entar auf zwei Zeilen
```

werden also bei der Suche nach »Kommentar « nicht gefunden. Wenn der Schalter -r (oder -r+) angegeben ist, dann behandelt GREP den *Suchstring* als »Ausdruck« und interpretiert die folgenden Zeichen auf spezielle Weise:

^ steht für den Beginn einer Zeile, wenn es als erstes Zeichen des Suchstrings angegeben ist.

$ steht für ein Zeilenende, wenn es als letztes Zeichen des Suchstrings angegeben ist.

. steht für ein beliebiges Zeichen.

***** steht für eine beliebige Anzahl von Wiederholungen des vorangehenden Zeichens: die Suche nach fo* findet f (0 Wiederholungen von o), fo, foo, fooo usw.

+ arbeitet ähnlich wie *, setzt aber mindestens ein Vorkommen des vorangehenden Zeichens voraus: die Suche nach fo+ findet fo, foo, fooo usw. – nicht aber f.

**** behandelt das nächste Zeichen direkt und ermöglicht die Suche nach Zeichen, die ansonsten speziell interpretiert würden: \. sucht auch in einem Ausdruck nach einem Dezimalpunkt, \$ nach einem Dollarzeichen usw.

[] ein durch eckige Klammern eingeschlossener String wird als *Zeichensatz* behandelt – GREP sucht (unabhängig von der Reihenfolge der Angabe) nach sämtlichen Zeichen, die in diesem String enthalten sind. Mit zwei Zeichen und einem Bindestrich kann ein *Zeichenbereich* angegeben werden: [*a-z*] sucht z.B. nach sämtlichen Kleinbuchstaben. Mehrere Bereiche lassen sich kombinieren: [*a-bd-z*] sucht nach sämtlichen Kleinbuchstaben außer *c*.

Ein vorangestelltes Caret (^) wird bei Zeichensätzen als Negation interpretiert: eine Angabe wie [*xyz*] sucht nach den Buchstaben x, y und z; die Angabe [^*xyz*] nach sämtlichen Zeichen *außer* x, y und z. Das Caret läßt sich auch auf Zeichenbereiche anwenden: [^*a-f*] sucht nach sämtlichen Zeichen außer den Kleinbuchstaben von a bis f, [^*a-bd-z?*] zusammen mit dem Schalter -i nur nach Sonderzeichen außer dem Fragezeichen und dem Buchstaben c.

Hinweis: Die Spezialzeichen werden innerhalb eines Zeichensatzes nicht gesondert interpretiert; das Caret (^) wird nur dann als Negation gewertet, wenn es direkt am Anfang des Zeichensatzes steht (also direkt nach dem [).

Alle in dieser Liste nicht aufgeführten Zeichen werden auch bei Verwendung des Schalters -r direkt verglichen. Ausdrücke lassen sich beliebig miteinander verbinden (siehe nächster Abschnitt).

Einige Beispiele

In den folgenden Beispielen gehen wir davon aus, daß die Standardvorgabe für sämtliche Schalter auf »-« gesetzt ist (d.h. Sie die von uns gelieferte Version verwenden).

Aufruf `grep Cursor( *.pas`

Findet
```
Cursor(
GetCursor(
```

Findet nicht
```
GetCursorSize(...
CURSOR(
```

Sucht in allen Dateien mit der Endung .PAS im momentan gesetzten Verzeichnis.

Hinweis Standardmäßig wird zwischen Groß- und Kleinschreibung unterschieden.

Aufruf `grep -r [^a-z]cursor\ *( *.pas`

Findet
```
cursor(i:Integer): Integer;
cursor    (i,j,:Integer): Integer;
if (Cursor(10,12) then
```

Findet nicht
```
Getcursor(...
CURSOR(....
```

Sucht in allen Dateien mit der Endung .PAS im momentan gesetzten Verzeichnis

Hinweis Der Suchstring *cursor* enthält hier ein Leerzeichen, das wegen des umgekehrten Schrägstrichs (\) nicht als Trennung zweier Kommandozeilen-Parameter betrachtet wird – das Einschließen des gesamten Ausdrucks [^*a-z*]*cursor* *(in Anführungszeichen hätte denselben Effekt. Das Sternchen nach dem Leerzeichen steht für eine beliebige Anzahl von Wiederholungen. Über den Zeichensatz [^*a-z*] werden sämtliche Buchstaben vor *cursor* von der Suche ausgeschlossen (Leerzeichen und alle Sonderzeichen dagegen nicht).

Aufruf `grep -ri [a-c]:\\data.fil *.pas *.inc`

Findet
```
A:\data.fil
C:\DATA.FIL
b:\Data.FIL
```

Findet nicht
```
d:\data.fil
```

```
a:data.fil
A:\\DATA.FIL\\
```

Sucht in allen Dateien des momentan gesetzten Verzeichnisses, die das Suffix .PAS oder .INC haben.

Hinweis Ein Zeichen, das auf einen umgekehrten Schrägstrich folgt, wird von GREP direkt verwendet. Das gilt natürlich auch für den umgekehrten Schrägstrich selbst.

Aufruf

```
grep -ri [^a-z]Wort[^a-z] *.doc
```

Findet

```
jedes Wort muß auf einer eigenen Zeile stehen.
Zuerst ein Wort, dann ein Nach-Wort ...
```

Findet nicht

```
Mehrere Worte auf einer Zeile sind nicht erlaubt.
Nachwort schreibt man richtig so!
```

Sucht in allen Dateien des momentan gesetzten Verzeichnisses, die das Suffix .DOC haben.

Hinweis Die Suche nach »Wort« entspricht den Optionen *uw* des Editors von Turbo Pascal.

Aufruf

```
grep "Suchstring mit Leerzeichen" *.doc D:\DOKU\TP.DOC
```

Findet

```
Dies ist ein Suchstring mit Leerzeichen darin.
```

Findet nicht

```
Ein Suchstring mit   Leerzeichen
Ein SUCHSTRING mit Leerzeichen
```

Sucht in allen Dateien mit dem Suffix DOC innerhalb des momentan gesetzten Verzeichnisses und innerhalb der Datei TP.DOC, die auf dem Laufwerk D im Verzeichnis DOKU gespeichert ist.

Aufruf

```
grep -rd "[,.:?'\"]"$ \*.doc
```

Findet

```
Er sagte Hallo zu mir.
Wie gehts?
Vorausgesetzt, daß wir voraussetzen,
Ein Beispiel dazu:
"Manche Leute denken, Computer aber nicht."
```

Findet nicht

```
Er sagte "Hallo" zu mir.
Wie gehts? So lala, sprach er
```

Sucht in allen Dateien mit dem Suffix *.DOC des momentan gesetzten Verzeichnisses sowie aller Unterverzeichnisse, die in diesem Verzeichnis enthalten sind.

Hinweis Gesucht wird nach Komma, Punkt, Doppelpunkt, Fragezeichen und Anführungszeichen am Ende einer Zeile. Bei der

Angabe des Zeichensatzes muß das " mit einem umgekehrten Schrägstrich eingeleitet werden – ansonsten würde es als Ende des Suchstrings behandelt. Das *$* (Zeilenende) steht hier außerhalb des in Anführungszeichen eingefaßten Teils: GREP verbindet beide Angaben automatisch zu *einem* Suchstring.

Aufruf
```
grep -ild " der " \*.doc
grep -i -l -d " der " \*.doc
```

Findet
```
unter Berücksichtigung der Tatsache, daß
... ärgerlich. Der Hauptgrund ...
Wieso? Ganz einfach ä der Compiler
```

Findet nicht
```
Derselbe Fehler wie oben
Geht nicht, weil das gesuchte Begriff fehlt
```

Sucht in — allen Dateien mit dem Suffix .DOC – und zwar in sämtlichen Verzeichnissen des Laufwerks, weil das Stammverzeichnis als Suchweg zusammen mit dem Schalter `-d`angegeben ist.

Hinweis — Offensichtlich spielt es keine Rolle, in welcher Reihenfolge und wie die einzelnen Schalter angegeben werden.

Aufruf
```
grep -rw Wort *.doc
```

Findet
```
Also sprach der Herr: Am Anfang war das WORT.
Ein Grußwort, eigentlich falsch erwischt
Zuerst ein Wort, dann ein Nach-Wort ...
```

Findet nicht
```
Mehrere Worte auf einer Zeile sind nicht erlaubt.
Nachwort schreibt man richtig so!
```

Sucht in — allen Dateien des momentan gesetzten Verzeichnisses, die das Suffix .DOC haben.

Hinweis — Weil GREP das »ß« standardmäßig als Trennzeichen behandelt, wird die Zeichenfolge »wort« in »Grußwort« ebenfalls als eigenständiges Wort betrachtet.

Aufruf
```
grep -w[=] = *.pas
```

Findet
```
if i = 5 then ...
j := 5;
```

Findet nicht
```
{ ==================== }
```

Sucht in — allen Dateien des momentan gesetzten Verzeichnisses, die das Suffix .PAS haben.

Hinweis — Hier wurde der Zeichensatz für »Trennzeichen« neu definiert und dann nach ganzen Wörtern gesucht: GREP findet sämt-

liche Vergleiche und Zuordnungen in einem Pascal-Quelltext, ignoriert aber zur optischen Trennung eingezogene Zeilen.

MAKE.EXE – die automatische Verwaltung von Programmen

Die grundlegenden Details und der Zweck einer automatischen Verwaltung von Programmteilen wurden bereits in Kapitel 5 »Verwaltung von Projekten« besprochen; wir konzentrieren uns in diesem Kapitel deshalb auf die technischen Details.

Die Erzeugung von MAKE-Dateien

Dateien für MAKE können in beliebiger Zahl mit jedem ASCII-Editor erzeugt werden. Sämtliche Regeln, Definitionen und Direktiven werden mit einem Zeilenvorschub beendet; ein umgekehrter Schrägstrich (\) als letztes Zeichen einer Zeile dient nötigenfalls als »Verlängerung«:

```
Testprog.exe: testprog.pas, test_1.inc, test_2.inc, util.tpu, \
  screenio.tpu, meier.pas, müller.pas, hinz.pas, kunz.pas
```

Bezeichner, Parameter und Befehle werden durch Leerzeichen und/oder Tabs voneinander getrennt, Zeilen mit Befehlen können bzw. müssen durch Leerzeichen und/oder Tabs eingerückt werden (siehe übernächster Abschnitt). Leerzeilen sind an beliebiger Stelle zulässig und werden ignoriert.

Der Dateiname MAKEFILE ist lediglich eine Standardvorgabe und wird nur dann benutzt, wenn Sie MAKE ohne den Kommandozeilen-Parameter *-f* aufrufen. Ein Aufruf wie

```
make -fmyprog
```

sucht die Datei MYPROG (ohne Suffix!) im momentan gesetzten Verzeichnis und interpretiert ihren Inhalt als Abhängigkeitsdefinitionen.

Eine MAKE-Datei sieht einem Programm sehr ähnlich und setzt sich aus den folgenden Elementen zusammen:

- Kommentaren
- expliziten Regeln
- impliziten Regeln
- Makro-Definitionen
- Direktiven (Aufnahme von Dateien, bedingte Ausführung, Fehlererkennung und »Undefinition« von Makros).

Die folgenden Abschnitte beschäftigen sich mit diesen Oberpunkten im Detail.

Kommentare

werden mit einem Doppelkreuz (#) eingeleitet. MAKE ignoriert sämtliche Zeichen zwischen einem # und dem nächsten Zeilenvorschub. Spezielle Voraussetzungen hinsichtlich der Position von Kommentaren innerhalb einer Zeile gibt es nicht – aber eine Reihe anderer Grenzen, auf die Sie achten müssen:

Ein umgekehrter Schrägstrich kann **nicht** zur Fortsetzung eines Kommentars in die nächste Zeile verwendet werden, jede Kommentarzeile muß mit einem eigenen # beginnen. Zeilen mit Kommentaren können prinzipiell nicht verlängert werden: wenn sich der umgekehrte Schrägstrich vor dem Doppelkreuz befindet, ist er nicht das letzte Zeichen der Zeile; befindet er sich dahinter, wird er als Teil des Kommentars ignoriert.

Einige Beispiele:

```
# MAKE-Datei für STERNE.EXE
# übernimmt die Verwaltung für das gesamte Projekt
# Implizite Regel:
.asm.obj                # .OBJ-Dateien hängen von .ASM-Dateien ab
  tasm $*.asm,$*.obj;   # TASM- (oder MASM-)Aufruf zur Erzeugung
# unbedingte Regel ä STERNE.EXE wird immer erzeugt:
sterne.exe:
  tpc sterne /m         # TPC-Aufruf zur Compilierung
# Abhängigkeiten:
stlib2.obj: stlib2.asm  # verwendet die implizite Regel (s.o.)

# ZWISCHEN den nächsten beiden Zeilen darf kein Kommentar stehen!
stlib1.obj: stlib1.asm  # eine eigene explizite Regel:
  tasm stlib1.asm,stlib1.obj;
# Ende der MAKE-Datei
```

Explizite Regeln und Abhängigkeiten

haben wir in Kapitel 5 »Verwaltung von Projekten« bereits ausführlich demonstriert, ohne uns groß um Formales zu kümmern. Eine explizite Abhängigkeit wird folgendermaßen formuliert:

```
Ziel [Ziel ...]: [Quelle Quelle ...]
  [Befehl]
  [Befehl]
  ...
```

Ziel steht hier für den Namen der zu überprüfenden Datei, *Quelle* für die Datei, von der Ziel abhängig ist; *Befehl* kann eine beliebige DOS-Kommandozeile sein, die auch den Aufruf anderer Programme und/oder .BAT-Dateien enthalten darf.

Explizite Abhängigkeiten definieren eine oder mehrere Ziel-Dateien, eine (optionale) Anzahl von Quellen und eine (ebenfalls optionale) Anzahl von DOS-Kommandozeilen. Die Angabe von Laufwerksbezeichnern und Suchwegen für *Ziel* und/oder *Quelle* ist zulässig, Jokerzeichen sind es nicht.

Definitionen, die nur aus *Ziel* [*Ziel ...*]: bestehen, sind syntaktisch legal, werden aber auf spezielle Weise behandelt (siehe nächster Abschnitt). Eine explizite Abhängigkeitsdefinition hat den folgenden logischen Hintergrund:

- MAKE prüft zuerst, ob die als *Ziel* angegebene Datei existiert. Wenn nicht, werden die folgenden *Befehle* ausgeführt, die letztendlich die Erzeugung von *Ziel* zur Folge haben sollten. (Natürlich hält Sie niemand davon ab, auf das Fehlen einer Datei mit der Ausgabe eines Verzeichnis-Listings – und sonst weiter gar nicht – zu reagieren).

- Wenn *Ziel* bereits existiert, findet ein Uhrzeit- und Datumsvergleich zwischen *Ziel* und den angegebenen *Quellen* statt. Falls auch nur eine der *Quellen* neueren Datums als *Ziel* ist, werden die *Befehle* ebenfalls ausgeführt. (Hier gilt natürlich auch, daß *Ziel* und *Quelle* in irgendeinem logischen Zusammenhang stehen sollten – zwingend ist das allerdings nicht).

- Die Prüfung des Datums von *Quelle* enthält ihrerseits eine Prüfung, ob diese Datei bereits existiert bzw. ob für sie ebenfalls eine Abhängigkeitsdefinition vorhanden ist. Wenn ja, dann wird diese Definition zuerst bearbeitet (und *Quelle* auf den neuesten Stand gebracht).

Eine Definition gegenseitiger Abhängigkeiten (Datei A hängt von B ab; Datei B hängt von C und D ab usw.) kann als Baumstruktur aufgefaßt werden, bei der die Bearbeitung der »Zweige« als erstes stattfindet. *Ziel*-Angaben stellen Knotenpunkte dar. Jeder Dateiname darf folglich innerhalb der gesamten MAKE-Datei nur einmal als *Ziel* erscheinen.

Hinweis: MAKE liest als ersten Schritt die MAKE-Datei ein und stellt eine Baumstruktur her – in welcher Reihenfolge explizite Regeln im Text erscheinen, spielt keine Rolle.

Die Formatierung der Zeilen ist sowohl in Bezug auf die Reihenfolge als auch auf die Einrückung zwingend vorgeschrieben: *Ziel* muß in der ersten Spalte beginnen, *Befehl* muß mindestens um ein Leerzeichen eingerückt sein. Sowohl für die Formulierung der Abhängigkeit als auch für die Angabe der Befehle kann ein umgekehrter Schrägstrich als Verlängerung der jeweiligen Zeile verwendet werden.

MAKE betrachtet jede Zeile, die auf eine Regel folgt und mindestens um ein Leerzeichen eingerückt ist, als Teil der Kommandoliste für diese Regel. Im folgenden Beispiel werden also drei DOS-Kommandos hintereinander ausgeführt, wenn der erste Test fehlschlägt:

```
<Regel (Abhängigkeitsdefinition)>
  DOS-Befehl 1
  DOS-Befehl 2  # dieser Kommentar ist erlaubt
  DOS-Befehl 3
# diese Zeile beginnt wieder in Spalte 1
```

Was dieses Fragment zusätzlich verdeutlichen soll: Kommentarzeilen innerhalb einer Kommandoliste sind nicht erlaubt – wenn sie mit der ersten Spalte beginnen, werden sie als Ende der Kommandoliste interpretiert, bei eingerückten Kommentaren wie

```
< Regel >
  # ein FALSCH plazierter Kommentar
  DOS-Befehl 1
```

würde MAKE versuchen, die Kommentarzeile als DOS-Kommandozeile auszugeben. Leerzeilen innerhalb der Kommandoliste werden dagegen ignoriert.

Abhängigkeitsdefinitionen ohne folgende Kommandozeilen

werden von MAKE in unterschiedlicher Weise behandelt – *Ziel*-Dateien, die in expliziten Regeln ohne folgende Kommandozeilen angegeben sind, hängen nicht nur von den *Quellen* ab, sondern auch von jeder Datei, für die eine *implizite* Regel zutrifft (siehe nächsten Abschnitt).

Einige Beispiele für explizite Regeln

```
myutil.obj:  myutil.asm
  masm myutil.asm,myutil.obj;

myapp.exe:  myapp.pas myglobal.tpu myutils.tpu
  tpc myapp /Tc:\tp\bin

myglobal.tpu:  myglobal.pas
  tpc myglobal /Tc:\tp\bin

myutils.tpu:  myutils.pas  myglobal.tpu  myutil.obj
  tpc myutils /Tc:\tp\bin
```

Diese vier Regeln geben folgende Zusammenhänge wieder:

- MYUTIL.OBJ hängt von MYUTIL.ASM ab – wenn MYUTIL.ASM seit der letzten Compilierung verändert wurde, wird MYUTIL.OBJ über den angegebenen Aufruf von MASM neu erzeugt.

- Die zweite Regel legt fest, daß MYAPP.EXE von MYAPP.PAS, MYGLOBAL.TPU und MYUTILS.TPU abhängig ist. Wenn sich eine dieser drei Dateien verändert hat, dann wird MYAPP.EXE über den Aufruf von TPC neu erzeugt (auf den Kommandozeilen-Parameter */T* kommen wir gleich noch zurück).

- Die dritte Regel ist der ersten ausgesprochen ähnlich, nur daß hier der Zusammenhang zwischen einer Unit und einem Pascal-Quelltext definiert ist: wenn sich MYGLOBAL.PAS seit der letzten Compilierung verändert hat, dann wird MYGLOBAL.TPU über einen Aufruf von TPC neu erzeugt.

- Die letzte Regel legt fest, daß MYUTILS.TPU von MYUTILS.PAS, MYGLOBAL.TPU und MYUTIL.OBJ abhängig ist. Wenn sich auch nur eine dieser drei Dateien seit der letzten Compilierung verändert hat, wird TPC entsprechend aufgerufen und bringt diese Unit auf den neuesten Stand.

So weit, so gut – allerdings läßt sich das gezeigte Beispiel in einigen Punkten durchaus verbessern:

- Ein Aufruf von MAKE ohne Angabe eines *Ziel*-Dateinamens verwendet das Ziel der ersten expliziten Definition – die oben als zweites gezeigte Regel sollte also eigentlich an erster Stelle stehen.

- MAKE verwendet implizite Regeln für alle Dateien, bei denen keine explizite Regel definiert ist, »weiß« also von selber, daß eine Datei wie MYGLOBAL.TPU von dem Quelltext MYGLOBAL.PAS abhängig ist. Die Prüfung von Datum und

Uhrzeit können wir den Compiler (über den Kommandozeilen-Parameter /M) erledigen lassen.

Unter Berücksichtigung dieser beiden Punkte verkürzt sich das oben gezeigte Beispiel auf:

```
myapp.exe: myapp.pas myglobal.tpu myutils.tpu myutil.obj
  tpc myapp /M /Tc:\tp\bin

myutil.obj: myutil.asm
  masm myutil.asm,myutil.obj
```

(Die Liste der Dateien für MYAPP.EXE ist hier etwas länger geworden – wir haben einfach die Dateien, von denen MYUTILS.TPU abhängig ist, mit aufgenommen).

Implizite Regeln und Abhängigkeiten

sind eine Erweiterung expliziter Regeln – sie gehen davon aus, daß eine Datei des Namens xxx.TPU oder xxx.EXE von einem Quelltext desselben Namens (nur eben mit der Endung .PAS) abhängig ist, gelten aber für *alle* Dateien. Die beiden folgenden Definitionen haben für die Datei STERNLIB dieselbe Wirkung; die erste ist explizit, die zweite implizit formuliert:

```
myutil.obj: myutil.asm
  masm myutil.asm,myutil.obj;

.asm.obj:
  masm $*.asm,$*.obj
```

Diese Definition besagt: *Sämtliche* .OBJ-Dateien werden gegen ihre Quelltexte geprüft – wenn der jeweilige Quelltext neueren Datums als die Object-Datei ist, folgt ein Aufruf des Assemblers. (Wofür *$** steht? Für ein spezielles Makro zur Repräsentation der Dateinamen, es ist im nächsten Abschnitt besprochen). Die allgemeine Syntax einer impliziten Regel ist

```
.Quell-Suffix.Ziel-Suffix:
  [Befehl]
  [Befehl]
  ...
```

Wie auch bei expliziten Regeln sind *Befehle* optional und müssen mindestens mit einem Leerzeichen eingerückt werden. Der dem *Quell-Suffix* vorangestellte Punkt muß in der ersten Spalte der jeweiligen Zeile stehen.

MAKE stellt den angegebenen Suffixen automatisch einen Dateinamen voran; eine implizite Regel des Formats

```
.xxx.yyy
  DOS-Befehl 1
  DOS-Befehl 2
  ...
```

ersetzt also sämtliche expliziten Regeln der Art

```
Dateiname.xxx: Dateiname.yyy
```

```
DOS-Befehl 1
DOS-Befehl 2
...
```

Der springende Punkt: **MAKE verwendet implizite Regeln automatisch**, wenn entweder

- für einen angegebenen Dateinamen keine explizite Regel existiert, oder
- für einen Dateinamen eine explizite Regel ohne folgende DOS-Kommandozeilen angegeben ist.

Einige Beispiele dazu, in denen wir allerdings nicht ohne einen weiteren Vorgriff auskommen:

```
.asm.obj:
  masm $*.asm,$*.obj;
```

Wenn wir davon ausgehen, daß diese Definition in der Datei MAKEFILE gespeichert ist, dann bewirkt der Aufruf

```
make demo.obj
```

den Vergleich zwischen den Dateien DEMO.ASM und DEMO.OBJ. Falls DEMO.ASM neueren Datums ist, wird DEMO.OBJ durch den Aufruf

```
masm demo.asm,demo.obj;
```

neu compiliert. (Hier wird der Kommandozeilen-Parameter DEMO.OBJ als Dateiname verwendet – MAKE bearbeitet also nicht sämtliche Object-Dateien im momentan gesetzten Verzeichnis.)

Wenn wir eine implizite Regel für die Erzeugung von Units an den Beginn der MAKE-Datei setzen, dann lassen sich die beiden im letzten Abschnitt gezeigten expliziten Regeln erheblich verkürzen:

```
.pas.tpu:
  tpc $<

myglobal.tpu: myglobal.pas
myutils.tpu: myutils.pas myglobal.tpu myutil.obj
```

Da wir hier keine explizite Angabe machen, mit welchen Befehlen die jeweilige .TPU-Datei erzeugt werden soll, verwendet MAKE die zuvor angegebene implizite Regel.

Weitere Formalitäten Eine MAKE-Datei kann mehrere implizite Regeln enthalten, die sich auf dasselbe *Ziel-Suffix* beziehen. Regeln werden in der Reihenfolge bearbeitet, in der sie im Text der MAKE-Datei erscheinen: nur die erste anwendbare Regel wird jeweils berücksichtigt.

MAKE berücksichtigt immer nur die erste zutreffende Regel für die Bearbeitung einer Datei. Das heißt: Selbst wenn sich im zuvor gezeigten Beispiel beim Aufruf des Compilers ein Fehler ergibt, findet danach keine Suche nach einer .ASM-Datei gleichen Namens statt.

Hinweis: Wenn sich mehr als eine implizite Regel auf ein *Ziel* bezieht, ist die

Reihenfolge der Regeln ausschlaggebend – ansonsten spielt sie (wie bei expliziten Regeln) keine Rolle.

Wie auch bei expliziten Regeln werden sämtliche Zeilen, die auf eine implizite Regel folgen und mit einem Leerzeichen (oder Tab) beginnen, als Kommandoliste interpretiert. Die Kommandoliste endet entweder mit der nächsten Zeile, deren erstes Zeichen kein Leerzeichen oder Tabulator ist, oder mit dem Ende der MAKE-Datei. Kommentarzeilen innerhalb einer Kommandoliste sind nicht zulässig.

Implizite Regeln enthalten grundsätzlich nur Suffixe. Dateinamen werden durch spezielle Makros ersetzt, deren Besprechung Sie im übernächsten Abschnitt finden.

Kommandolisten

können eine (fast) beliebige Länge haben, dürfen nicht durch Kommentarzeilen unterbrochen werden und müssen jeweils mit einem oder mehreren Leerzeichen eingerückt werden. Soviel haben wir bereits besprochen – eine formale Definition (und einige weitere Möglichkeiten) stehen dagegen noch aus. Jede Zeile einer Kommandoliste hat das folgende Format:

```
[Präfix ...] Befehl
```

Mit dem (optionalen) *Präfix* wird festgelegt, in welcher Art MAKE den folgenden *Befehl* behandelt. Drei Möglichkeiten sind definiert:

`@` unterdrückt die Darstellung des Befehls auf dem Bildschirm. Normalerweise gibt MAKE jede Kommandozeile vor ihrer Ausführung aus und ermöglicht so die Verfolgung des Ablaufs. `@` unterdrückt lediglich die Ausgabe der damit eingeleiteten Zeile (weitere Zeilen werden wieder auf dem Bildschirm dargestellt). Mit dem Parameter *-s* läßt sich die Ausgabe sämtlicher Kommandozeilen unterdrücken.

`-nnn` legt die Reaktion von MAKE auf Fehler bei der Ausführung des folgenden Befehls fest. *nnn* steht für eine Nummer im Bereich von 1 bis 255 und muß unmittelbar auf das »-« folgen. Wenn der durch den DOS-Befehl zurückgelieferte Exit-Code den angegebenen Wert übersteigt, bricht MAKE ab und löscht die *Ziel-Datei* der momentan aktiven Regel (soweit vorhanden). Bei einer Angabe wie

```
-4 myprog daten.x
```

setzt MAKE die Arbeit nach dem Aufruf des Programms MYPROG nur dann fort, wenn der von MYPROG zurückgelieferte Exit-Code zwischen 0 und 3 liegt.

Bei Befehlen, die ohne das Präfix *-nnn* gegeben werden, bricht MAKE die Arbeit ab, wenn sich ein Exit-Code ungleich 0 ergibt.

`-` (ohne eine folgende Nummer) legt fest, daß MAKE den von einem DOS-Befehl zurückgelieferten Exit-Code überhaupt nicht prüft und die Arbeit auf jeden Fall fortsetzt.

DOS-Befehle werden von MAKE exakt in derselben Weise behandelt, als wären sie auf der DOS-Kommandoebene eingegeben worden – wenn man von

der Erweiterung spezieller Makros und der Tatsache absieht, daß die Umleitung der Ein/Ausgabe mit > und < sowie das »Pipelining« (mit |) nicht unterstützt wird. Die folgenden Befehle werden durch Laden von COMMAND.COM ausgeführt:

```
break   cd      chdir   cls     copy
ctty    date    del     dir     erase
md      mkdir   path    prompt  ren
rename  set     time    type    ver
verify  vol
```

Alle anderen Befehle führen den normalen Suchalgorithmus von MS-DOS aus:

- Absuche des momentan gesetzten Verzeichnisses, danach Absuche sämtlicher Verzeichnisse in der durch PATH vorgegebenen Folge;
- in jedem Verzeichnis wird der angegebene Name zuerst als .COM-, dann als .EXE-, und schließlich als .BAT-Datei gesucht;
- wenn es sich um eine .BAT-Datei handelt, wird COMMAND.COM zur Ausführung in den Speicher geladen.

(Wenn Sie in einem DOS-Befehl explizit ein Suffix angeben, wird natürlich nur nach einer Datei mit diesem Suffix gesucht.)

Einige Beispiele Der folgende DOS-Befehl wird über das Laden von COMMAND.COM ausgeführt:

```
cd c:\tp\include
```

Ein Aufruf von TPC bedingt normalerweise die Ausführung einer vollständigen Suche:

```
tpc myprog.pas /$B+,R+,I+
```

Die explizite Angabe des Suffix .COM führt ebenfalls eine Suche in sämtlichen Verzeichnissen aus, läßt aber .EXE- und .BAT-Dateien unberücksichtigt:

```
myprog.com daten.xyz
```

Und schließlich: bei der Angabe eines vollständigen Suchwegs und eines expliziten Suffix unterbleibt jede automatische Suche:

```
c:\util\textsort.com demo.xrf
c:\tp\tpc.exe myprog.pas /$B+,R+,I-
```

Makros

kennt MAKE in mehreren Formen: als einfachen Ersatz für Text, als vordefiniert oder für spezielle Zwecke. Fangen wir mit der einfachsten Art und einem Beispiel an:

In einer MAKE-Datei erscheinen bestimmte Dateinamen, Kommandozeilen-Parameter und andere Dinge meist mehr als einmal (so wie der Parameter `/Tc:\tp\bin`, den wir bei fast allen vorangegangenen Beispielen verwendet

haben – er gibt an, daß die Dateien TURBO.TPL und TPC.CFG im Verzeichnis C:\TP\BIN gesucht werden sollen). Falls Sie zu einem späteren Zeitpunkt eine andere Konfigurationsdatei verwenden wollen (oder müssen), gibt es zwei Möglichkeiten: entweder korrigieren Sie an einem Dutzend verschiedener Stellen in der MAKE-Datei oder Sie verwenden von vornherein ein Makro, dessen Definition hier recht einfach ausfällt:

```
TURBO=c:\tp\bin
```

Danach können wir unsere MAKE-Datei folgendermaßen umschreiben:

```
# Makro-Definition
TURBO=c:\tp\bin

# implizite Regel für .TPU-Dateien
.pas.tpu:
  tpc $< /T$(TURBO)
# implizite Regel für .OBJ-Dateien
.asm.obj:
  masm $<.asm,$<.obj;

myapp.exe: myapp.pas myglobal.tpu myutils.tpu
  tpc myapp /T$(TURBO)

myutils.tpu: myutils.pas myglobal.tpu myutil.obj

# myglobal.pas -> myglobal.tpu und
# myutil.asm -> myutil.obj
# werden durch implizite Regeln bearbeitet!
```

MAKE ersetzt hier jedes Vorkommen von $(TURBO) automatisch durch die Zeichenfolge *c:\tp\bin*. Die Vorteile dürften auf der Hand liegen – um das gesamte Programm mit einer anderen Konfigurationsdatei zu compilieren, muß lediglich eine einzige Definition geändert werden.

Eine weitere Möglichkeit besteht darin, den aktuellen Wert des Makros TURBO erst beim Aufruf von MAKE festzulegen. Innerhalb der MAKE-Datei findet keine Definition von TURBO statt, MAKE wird jeweils mit dem Kommandozeilen-Parameter *-D* (define macro) aufgerufen. Ein Aufruf wie

```
make -DTURBO=c:\tp\bin
```

weist TURBO den Wert *c:\tp\bin* zu. (Auf weitere Kommandozeilen-Parameter kommen wir noch zurück).

Die Definition eines Makros in der MAKE-Datei hat die folgende Form:

```
Makroname  =  Text
```

Makroname folgt den üblichen Regeln für Pascal-Bezeichner (beginnt also mit einem Buchstaben, enthält nur Buchstaben und Ziffern, keine Umlaute), darauf folgen eine beliebige Anzahl von »weißen Leerzeichen« (oder auch überhaupt keines) und ein Gleichheitszeichen. Führende Leerzeichen von *Text* werden ignoriert, die restlichen Zeichen zwischen dem Gleichheitszeichen und dem

nächsten Zeilenvorschub werden als Wert des Makros interpretiert. Der *Text* ersetzt jedes Vorkommen des *Makronamens* innerhalb der MAKE-Datei und kann beliebige Zeichenfolgen enthalten.

Wiederholte Definitionen ein und desselben Makros sind zulässig – MAKE bewertet jeweils die zuletzt gelesene Definition eines Makros als gültig. (Was auch bedeutet: ein zu Anfang der MAKE-Datei definiertes Makro kann über den Kommandozeilen-Parameter *-D* nicht verändert werden.)

Zwischen Groß- und Kleinschreibung wird unterschieden: *TURBO*, *Turbo* und *turbo* stellen also drei verschiedene Makros dar.

Achtung! Der Name eines Makros muß mit dem ersten Zeichen der Zeile beginnen, eine Einrückung ist hier nicht erlaubt.

Der Aufruf eines Makros geschieht über ein Dollarzeichen ($), gefolgt vom Namen des Makros in Klammern, also

```
$(Makroname)
```

MAKE ersetzt diese Zeichenfolge (inklusive des *$* und der Klammern) durch den Text, der dem entsprechenden Makro zugewiesen wurde. Wenn das Makro zu diesem Zeitpunkt nicht definiert ist, wird ein Nullstring (d.h. überhaupt nichts) eingesetzt.

Die Klammern um den Makronamen sind zwingend vorgeschrieben – auch dann, wenn der Makroname nur aus einem Zeichen besteht. (Vordefinierte Makros machen hier allerdings eine Ausnahme, siehe übernächster Abschnitt).

Spezielle Konstrukte

Makros innerhalb von Makros Ein Makro kann nur auf der rechten Seite einer Makrodefinition erscheinen (die Konstruktion von Makronamen über Makros ist nicht erlaubt). Er wird erst dann erweitert, wenn das durch ihn definierte Makro erweitert wird. Anders gesagt: Nach einer Definitionsfolge wie

```
Makro1 =Text1
MMM =$(Makro1)
   ...
Makro1 =Text2
   ... $(MMM)
```

ergibt die Erweiterung *$(MMM)* den Text *Text2* (und nicht *Text1*).

Makros in Regeln werden sofort erweitert, weil die Bewertung von Regeln nicht linear, sondern als Baumstruktur erfolgt. Hier sollte man mit Redefinitionen vorsichtig sein.

Makros in Direktiven werden ebenfalls sofort erweitert. Nicht definierte Makros zusammen mit *!if* und *!elif* ergeben den Wert 0 (*False*).

Makros in DOS-Befehlen werden erst zum Zeitpunkt der Befehlsausführung erweitert – gültig ist also die jeweils zuletzt eingelesene Definition.

Vordefinierte Makros müssen (im Gegensatz zu selbstdefinierten) nicht in Klammern angegeben werden. MAKE kennt insgesamt fünf davon:

$d – zur Prüfung, ob andere Makros definiert sind oder nicht;

$* – liefert einen Dateinamen ohne Suffix;

$< – liefert einen Dateinamen mit Suffix;

$: – liefert einen Suchweg ohne Dateinamen;

$. – liefert einen Dateinamen mit Suffix ohne Suchweg;

$& – liefert einen Dateinamen ohne Suchweg.

Zusätzlich ist das Makro *__MAKE__* während der Ausführung von MAKE immer als 1 definiert, sämtliche via SET definierten Strings werden automatisch geladen und können als vordefinierte Makros verwendet werden. Im einzelnen:

Das Prüfmakro $d liefert den Wert 1 (*True*) zurück, wenn das angegebene Makro definiert ist, ansonsten wird der Wert 0 (*False*) erzeugt. Der Inhalt des angegebenen Makros spielt keine Rolle – geprüft wird lediglich sein Vorhandensein. *$d* kann nur zusammen mit den Direktiven *!if* und *!elif* verwendet werden. Das folgende Beispiel verwendet automatisch die Konfigurationsdatei im Verzeichnis *c:\tp\bin*, wenn das entsprechende Makro nicht über einen Kommandozeilen-Parameter definiert wurde:

```
!if !$d(TURBO)     # wenn TURBO nicht definiert ist:
TURBO=c:\tp\bin    # Definition als c:\tp\bin
!endif
```

(Der Operator *!* vor *$d* stammt aus der Programmiersprache »C« und steht für eine Negation (NOT)). Beim Aufruf von MAKE (mit dieser MAKE-Datei) in der Form

```
MAKE -DTURBO=c:\tp\projects  <weitere Parameter>
```

ist das Symbol TURBO definiert – folglich wird es innerhalb der MAKE-Datei mit *c:\tp\projects* gleichgesetzt. Ein Aufruf ohne explizite Definition, also

```
MAKE  <weitere Parameter>
```

entspricht das Symbol TURBO dagegen innerhalb der MAKE-Datei der Zeichenfolge *c:\tp\bin* (sozusagen als Standardvorgabe).

Makros für Dateinamen bekommen innerhalb einer Regel immer den als erstes angegebenen Namen zugeordnet. Innerhalb einer expliziten Regel wie

```
stlib.exe: stlib.pas
  ...
```

liefern die Makros *$<* und *$.* also den Namen *stlib.exe* zurück. Bei einer impliziten Regel steht dagegen das *Quell-Suffix* an erster Stelle. Diese Definition

```
.pas.exe:
  ...
```

liefert bei ihrem Aufruf mit dem Dateinamen *xxx* über die Makros *$<* und *$.* den Namen *xxx.pas* zurück.

Das Makro $* liefert einen Dateinamen (inklusive eines eventuellen Suchwegs) ohne Suffix zurück; aus

```
A:\X\DEMO.XYZ
```

wird also

```
A:\X\DEMO
```

Die Zuordnung »Makro – Dateiname« ergibt sich bei Regeln implizit – das Makro enthält immer den als erstes angegebenen Namen. Der Aufruf von TPC für die Datei MYAPP.EXE ließe sich also folgendermaßen verkürzen:

```
myapp.exe: myapp.pas myglobal.tpu myutils.tpu
  tpc $* /T$(TURBO)
```

Innerhalb dieser Regel wird das Makro *$** jeweils auf den Namen der *Ziel*-Datei (also *MYAPP*) erweitert.

Entsprechendes gilt für implizite Regeln – hier wird dem Makro ebenfalls der Name der ersten Datei (also der *Quelle*!) zugeordnet. Wenn wir davon ausgehen, daß das Makro TURBO definiert ist, dann könnte eine implizite Definition so aussehen:

```
.pas.exe:
  tpc $*.PAS /T$(TURBO)
```

(Das Abschneiden des Suffix und das explizite Anhängen von .PAS ist in diesem Fall allerdings unnötig – MAKE sucht sowieso nur Dateien mit dem Suffix .PAS heraus, das Makro *$<* wäre also äquivalent.)

Das Makro $< liefert den vollständigen Dateinamen innerhalb einer Regel zurück, d.h. einen eventuell angegebenen Suchweg, den Dateinamen selbst und das Suffix. Aus

```
A:\X\DEMO.XYZ
```

wird also

```
A:\X\DEMO.XYZ
```

d.h. es findet überhaupt keine Veränderung statt. Die folgende Regel kopiert zuerst die alte Version von STLIB.OBJ in das Unterverzeichnis BACKUPS:

```
stlib.obj: stlib.pas
  ä copy $< \backups    # keine Rückmeldung (-)
  tpc $* /T$(TURBO)
```

Da Makros bei impliziten Definitionen ebenfalls den als ersten aufgeführten Dateinamen annehmen, läßt sich eine Regel wie

```
.asm.obj:
  masm $*.asm,$*.obj;
```

auch einfacher schreiben, nämlich als

```
.asm.obj:
  masm $<,$*.obj;
```

Das Makro $ liefert den Suchweg einer Datei ohne den Dateinamen selbst zurück. Aus

```
A:\X\DEMO.XYZ
```

wird also

```
A:\X\
```

Das Makro $. liefert den Dateinamen und sein Suffix ohne einen eventuell angegebenen Suchweg zurück. Aus dem zuvor gezeigten Beispiel wird also

```
DEMO.XYZ
```

Das Makro $& liefert schließlich nur einen Dateinamen zurück, ein eventuell angegebener Suchweg und das Suffix werden abgeschnitten. Das Ergebnis wäre in unserem Fall

```
DEMO
```

Direktiven

sind in MAKE-Dateien in ähnlicher Weise wie in Pascal-Quelltexten möglich – sie werden hier aber mit einem Ausrufezeichen (!) eingeleitet und nicht in geschweifte Klammern gestellt. Verfügbar sind:

`!include`	zur Aufnahme anderer MAKE-Dateien
`!if !else !elif !endif`	für bedingte Regeln und Befehle
`!error`	zur Ausgabe von Meldungen jeder Art
`!undef`	»undefiniert« ein Makro

Achtung: Das Ausrufezeichen muß das erste Zeichen der jeweiligen Zeile sein (Einrückung ist nicht erlaubt), der Name der Direktive muß unmittelbar auf das Ausrufezeichen folgen.

Die Direktive !include fügt eine angegebene Datei an der entsprechenden Stelle in den Text der MAKE-Datei ein und hat eine der beiden folgenden Formen:

```
!include "Dateiname"
!include <Dateiname>
```

Hinweis: für C-Programmierer: MAKE macht keinen Unterschied zwischen <> und "" – die Angaben *<Dateiname>* und *"Dateiname"* werden exakt gleich bearbeitet.

!include-Direktiven können bis in beliebige Tiefen verschachtelt werden: Eine via !*include* eingefügte Datei kann also ihrerseits *!include*-Direktiven enthalten.

Mehrfache Aufnahmen ein und derselben Datei sind nur auf derselben Ebene möglich. Eine Konstruktion wie

```
!include "text1"
   # Datei TEXT1 enthält:
     ...
     !include "text2"
```

```
# Datei TEXT2 enthält:
   ...
  !include "text1"
```

führt zu einer Fehlermeldung. Formaler: *!include*-Direktiven, die endlose Schleifen bilden würden (TEXT1 nimmt TEXT2 auf, in TEXT2 steht wieder die Aufnahme von TEXT1), werden von MAKE als fehlerhaft zurückgewiesen. *!include*-Direktiven lassen sich selbstverständlich auch bedingt formulieren:

```
!if !$d(TURBO)
!include "suchweg.mac"
!endif
```

Dieses Beispiel nimmt die Datei SUCHWEG.MAC in den Text auf, wenn das Makro TURBO zum gegebenen Zeitpunkt nicht definiert ist.

Die Direktiven !if, !else, !elif und !endif arbeiten ähnlich wie die bedingte Compilierung von Turbo Pascal mit {*$IF*}, {*$ELSE*} usw. – nur daß hier vollständige Ausdrücke bewertet werden können.

Das allgemeine Format ist:

```
!if <Ausdruck>
  [Zeilen]
!elif <Ausdruck>
  [Zeilen]
!elif <Ausdruck>
  [Zeilen]
  ...
!else
  [Zeilen]
!endif
```

Jede Gruppe bedingter Ausdrücke wird mit *!if* begonnen und mit *!endif* abgeschlossen. *!elif* und *!else* sind optional und können vollständig fehlen, pro Gruppe sind ein *!else* und beliebig viele *!elif*-Direktiven zugelassen, die zwischen dem *!if* und einem eventuellen *!else* stehen müssen. [*Zeilen*] steht für eine oder mehrere Zeilen des Quelltextes mit folgendem Inhalt:

- Makrodefinition(en)
- explizite oder implizite Regel(n)
- *!include*-Direktive(n)
- weitere *!if*-Gruppen (beliebig tief verschachtelt)
- *!error*-Direktive(n)
- *!undef*-Direktive(n)

Regeln und dazugehörige Kommandolisten müssen grundsätzlich innerhalb eines *!if*-Zweiges komplett enthalten sein, d.h. die Verwendung von Bedingungen innerhalb einer Regel ist nicht erlaubt.

Zu jeder mit *!if* eingeleiteten Gruppe gehört ein entsprechendes *!endif*, das

sich innerhalb derselben Datei befinden muß – *!include*-Dateien mit unvollständigen Bedingungen wie

```
!if <Ausdruck>
  ...
!else
  ...
   <Dateiende>  # !endif in der "Haupt-Datei"
```

sind also nicht zulässig – unabhängig vom Text der Datei, die dieses Fragment via *!include* aufnimmt.

Ausdrücke werden mit einer »C«-ähnlichen Syntax formuliert und von MAKE über Integerwerte mit 32 Bit berechnet. Der Wert 0 gilt als *False*, alle Werte ungleich 0 gelten als *True*. Numerische Werte können in dezimaler, oktaler und hexadezimaler Notation angegeben werden:

```
4536   # dezimale Konstante
0677   # oktale Konstante (mit "0" eingeleitet)
0x2F4D # hexadezimale Konstante (mit "0x" eingeleitet)
```

MAKE unterstützt die folgenden unären Operatoren in Ausdrücken:

- `-` Negation
- `~` bitweises Komplement
- `!` logisches NOT

Die folgenden binären Operatoren (aus der Programmiersprache »C«) stehen zur Verfügung:

- `+` Addition
- `-` Subtraktion
- `*` Multiplikation
- `/` Division
- `%` Modulo
- `>>` Rechtsschieben
- `<<` Linksschieben
- `&` bitweises AND
- `|` bitweises OR
- `^` bitweises EXOR (Antivalenz)
- `&&` logisches AND
- `||` logisches OR
- `>` größer als
- `<` kleiner als
- `>=` größer oder gleich
- `<=` kleiner oder gleich
- `==` Gleichheit
- `!=` Ungleichheit
- `?:` bedingte Ausführung

Der »ternäre« Operator *?:* ist ein spezielles »C«-Konstrukt und hat die Form:

```
<Ausdruck_0> ? <Ausdruck_1> : <Ausdruck_2>
```

MAKE bewertet zuerst *Ausdruck_0*: wenn dieser Ausdruck einen Wert ungleich 0 annimmt (d.h. *True* ist), wird der Wert von *Ausdruck_1* als Gesamtergebnis zurückgeliefert, ansonsten der Wert von *Ausdruck_2*.

Die Rangfolge der Operatoren ist dieselbe wie in »C«: Operatoren gleicher Rangstufe werden von *links nach rechts* bearbeitet (außer `?:` – hier findet die Auswertung von rechts nach links statt). Klammern können in beliebiger Weise zur Gruppierung von Operatoren und Operanden eingesetzt werden.

Makros können in jedem Ausdruck anstelle von Operanden erscheinen, das Spezialmakro *$d* ist ebenfalls zulässig (und liefert den Wert 1 bzw. 0). Der Gesamtausdruck wird erst nach Erweiterung sämtlicher Makros bewertet und muß dann eine zulässige Operation darstellen.

Einige Beispiele:

```
!if FNAME != MYPROG
!error Falsche MAKE-Datei!
!else
FCOUNT = 5
TURBO=c:\tp\bin
!endif

!if FCOUNT <= 8 && TURBO == c:\tp\bin
  ....
!endif
```

Die Direktive !error bricht die Ausführung von MAKE ab und gibt eine Meldung aus. Das Format ist

```
!error <beliebiger Text>
```

Zusammen mit `!if` lassen sich Bedingungen programmieren, unter denen MAKE die Arbeit beendet. Das folgende Beispiel prüft, ob das Makro TURBO definiert ist – wenn nicht, wird MAKE beendet:

```
!if !$d(TURBO)
!error Konfigurationsdatei undefiniert.
!endif
```

Falls TURBO zum Zeitpunkt der Bearbeitung dieser Bedingung nicht definiert ist, erhalten Sie eine Meldung wie

```
Fatal makefile 2: Error directive: Konfigurationsdatei undefiniert.
```

Hinweis: Der zusammen mit *!error* angegebene Text wird von MAKE nicht interpretiert, d.h. eine Erweiterung eventuell enthaltener Makros findet nicht statt.

Die Direktive !undef löscht die angegebene Makrodefinition. Falls das Makro zum entsprechenden Zeitpunkt nicht definiert ist, passiert überhaupt nichts. Das Format ist:

```
!undef Makroname
```

Der Aufruf von MAKE

MAKE kann mit einer ganzen Reihe von Parametern gestartet werden. Am einfachsten geschieht der Aufruf aber durch die DOS-Kommandozeile

```
make
```

MAKE sucht in diesem Fall die Datei MAKEFILE im momentan gesetzten Verzeichnis. Falls diese nicht existiert, wird nach MAKEFILE.MAK gesucht. Falls auch diese Datei nicht gefunden werden kann, bricht MAKE mit einer Fehlermeldung ab. Die allgemeine Syntax eines Aufrufs von MAKE ist:

```
make [Option Option ...] [Ziel-Datei Ziel-Datei ...]
```

Optionen können in beliebiger Folge angegeben werden und sind jeweils durch ein oder mehrere Leerzeichen voneinander getrennt. Über *Ziel-Datei* können ein oder mehrere Namen von Dateien angegeben werden, die (normalerweise) durch implizite Regeln bearbeitet werden sollen. Mehrere Angaben werden ebenfalls durch Leerzeichen voneinander getrennt und von MAKE in der Reihenfolge ihres Erscheinens bearbeitet. Wenn keine *Ziel-Datei* angegeben ist, beginnt MAKE mit dem Dateinamen, der als *Ziel* in der ersten expliziten Regel erscheint.

Hinweis: MAKE fügt angegebenen *Ziel-Dateien* nicht automatisch ein Suffix an – ein Aufruf wie

```
make demo
```

würde also versuchen, die Datei DEMO zu prüfen (und nicht DEMO.OBJ oder DEMO.EXE).

Sämtliche *Optionen* werden durch einen Bindestrich (-) eingeleitet. MAKE unterscheidet zwischen Groß- und Kleinschreibung (*-d* ist also nicht äquivalent zu *-D*). Die folgenden Optionen sind definiert:

-D*Makroname* definiert *Makroname* als »existierend« – jede Erweiterung dieses Makros ergibt ein einzelnes Leerzeichen.

-D*Makroname=String* definiert *Makroname* als »existierend« – jede Erweiterung dieses Makros ergibt den angegebenen *String*.

-I*Suchweg* legt fest, in welchen Verzeichnissen MAKE zusätzlich nach *!include*-Dateien suchen soll. Die erste Suche findet immer im momentan gesetzten Verzeichnis statt.

-U*Makroname* »undefiniert« ein Makro und hat nur im Zusammenhang mit BUILTINS.MAK eine Wirkung (siehe nächster Abschnitt).

-s unterdrückt die Ausgabe von DOS-Kommandozeilen auf den Bildschirm. Normalerweise wird jede nicht mit @ eingeleitete DOS-Kommandozeile vor ihrer Ausführung auf dem Bildschirm dargestellt.

-n wertet die MAKE-Datei aus und stellt die Kommandozeilen auf dem Bild-

schirm dar, ohne sie ausführen zu lassen – eine sehr nützliche Hilfe bei der Fehlersuche in MAKE-Dateien.

-f*Dateiname* verwendet die angegebene Datei als MAKE-Datei. Wenn diese Datei nicht gefunden werden kann und kein Suffix angegeben ist, sucht MAKE zusätzlich nach *Dateiname*.MAK.

-? oder **-h** gibt eine Liste sämtlicher definierten MAKE-Optionen aus. Die Angabe nicht definierter Optionen hat dieselbe Wirkung.

Einige Beispiele:

```
make -n -fsterne.mak
make -s
make -Iinclude -DTURBO=c:\tp\bin demo.exe
```

Der erste Aufruf von MAKE verwendet STERNE.MAK als MAKE-Datei und stellt die DOS-Kommandozeilen auf dem Bildschirm dar, ohne sie auszuführen.

Der zweite Aufruf verwendet MAKEFILE (oder MAKEFILE.MAK) als MAKE-Datei und unterdrückt die Darstellung der DOS-Kommandozeilen vor ihrer Ausführung.

Der dritte Aufruf verwendet ebenfalls MAKEFILE oder MAKEFILE.MAK. Mit *!include* aufgenommene MAKE-Dateien werden nicht nur im momentan gesetzten Verzeichnis, sondern auch im Unterverzeichnis INCLUDE gesucht, das Makro *TURBO* wird als *c:\tp\bin* definiert; die erste implizite Regel, die sich auf eine .EXE-Datei als *Ziel* bezieht, überprüft den Zusammenhang zwischen DEMO.EXE und dem dazugehörigen Quelltext (DEMO.PAS oder DEMO.ASM).

MAKE bricht die Arbeit ab, wenn...

das Ende der MAKE-Datei erreicht ist und alle Befehle bzw. Regeln ausgeführt wurden;

- die MAKE-Datei »fatale« Fehler enthält (Syntaxfehler führen lediglich zu einer Meldung, danach arbeitet MAKE weiter);
- ein ausgeführter DOS-Befehl einen Exit-Code ungleich 0 (bzw. einen höheren Exit-Code als mit *-nnn* angegeben) zurückliefert;
- die Ausführung eines DOS-Befehls oder MAKE selber durch die Eingabe von Ctrl C oder Ctrl Break unterbrochen werden.

Die Datei BUILTINS.MAK

Sie werden bald feststellen, daß bestimmte Makros und implizite Regeln praktisch für jedes MAKE-Projekt zur Anwendung kommen. Im Prinzip gäbe es hier drei Möglichkeiten:

- Sie geben die entsprechenden Texte jeweils neu ein;
- Sie fassen alle häufig benutzten Makros und Regeln in einer Datei zusammen, die jeweils via *!include* aufgenommen wird;
- Sie fassen Makros und Regeln in einer Datei namens BUILTINS.MAK zusammen und überlassen MAKE die (automatische) Aufnahme.

Etwas konkreter: MAKE sucht bei jedem Start zuallererst nach einer Datei mit dem Namen BUILTINS.MAK – falls diese Datei existiert, wird sie gelesen und interpretiert. Wenn BUILTINS.MAK vorhanden ist, sieht die Reihenfolge der Bewertung so aus:

1. BUILTINS.MAK

2. Optionen der Kommandozeile von MAKE. Hier lassen sich in BUILTINS.MAK enthaltene Makros verändern (mit *-D*) bzw. löschen (mit *-U*).

3. Die eigentliche MAKE-Datei (MAKEFILE bzw. die via *-f* angegebene Datei).

BUILTINS.MAK, MAKEFILE, !include und Suchwege

MAKE sucht BUILTINS.MAK zuerst im momentan gesetzten Verzeichnis, danach über sämtliche via PATH angegebenen Suchwege – diese allgemeinen Definitionen können also von mehreren Verzeichnissen aus zugänglich gemacht werden.

MAKEFILE bzw. die angegebene MAKE-Datei wird dagegen nur im momentan gesetzten Verzeichnis gesucht. Der Sinn dieser Begrenzung: In mehreren Verzeichnissen können verschiedene voneinander getrennte MAKE-Dateien gleichen Namens existieren, ohne daß MAKE sie durcheinander bringt.

Über *!include* aufgenommene Dateien werden ebenfalls innerhalb des momentan gesetzten Verzeichnisses gesucht: Wenn Sie die Option *-I* verwenden, findet die Suche zusätzlich im darüber angegebenen Verzeichnis statt.

Fehlermeldungen von MAKE

lassen sich in zwei Klassen unterteilen: »Fatal« und »Syntax«. Fehler der ersten Art haben einen sofortigen Abbruch zur Folge, Syntaxfehler erzeugen lediglich eine entsprechende Meldung, MAKE setzt die Interpretation der Regeln und Befehle danach fort.

Fehler mit sofortigem Abbruch »Fatal Errors«

Don't know how to make 'XXXXXXXX'

(Keine Ahnung, wie 'XXXXXXXX' erzeugt werden soll)
Eine Quelltext-Datei des angegebenen Namens existiert nicht, bzw. für die Erzeugung einer angegebenen Ziel-Datei ist weder eine explizite Regel vorhanden noch eine implizite Regel anwendbar.

Error directive: 'XXX'

(!error-Direktive: <Text>)
Wird ausgegeben, wenn MAKE eine *!error*-Direktive bearbeiten muß (d.h. diese Direktive nicht durch *!if* ausgeschlossen wurde).

Incorrect command line argument: 'XXX'
(Ungültiger Kommandozeilen-Parameter: 'XXX')

Not enough memory
(Nicht genügend Platz im Hauptspeicher)
Entfernen Sie speicherresidente Programme wie SideKick, verteilen Sie Ihre Quelltexte auf mehrere Module, unterteilen Sie die MAKE-Datei...

Unable to execute command
(DOS-Befehl kann nicht ausgeführt werden)
Mögliche Gründe sind: Tippfehler in der angegebenen Kommandoliste, fehlende Suchwege und – unwahrscheinlicher – Diskettenfehler oder Fehler innerhalb des aufgerufenen Programms selbst.

Unable to open makefile
(MAKE-Datei nicht gefunden)
Das momentan gesetzte Verzeichnis enthält keine Datei namens MAKEFILE oder MAKEFILE.MAK bzw. die via *-f* angegebene MAKE-Datei konnte nicht gefunden werden.

Fehler »Errors«

Bad file name format in include statement
(Ungültiger Dateiname in *!include*-Direktive)
Zusammen mit *!include* angegebene Dateinamen müssen entweder durch Anführungszeichen oder durch spitze Klammern eingeschlossen sein.

Bad undef statement syntax
(Unzulässige Anwendung von *!undef*)
Auf *!undef* darf nur der Name eines Makros innerhalb derselben Zeile folgen – nicht mehr und nicht weniger. Der Makroname wird direkt angegeben, d.h. ohne $ und/oder Klammern.

Character constant too long
(Zeichenkonstanten nur mit einem oder zwei Zeichen erlaubt)

Command arguments too long
(DOS-Kommandozeile hat mehr als 127 Zeichen)
Hier handelt es sich um eine von MS-DOS gesetzte Grenze.

Command syntax error
(Syntaxfehler)
Diese Meldung kann eine ganze Reihe von Ursachen haben:

- eine Regel, eine Direktive oder eine Makrodefinition beginnt nicht in der ersten Spalte der jeweiligen Zeile

- eine implizite Regel wurde nicht als *.xxx,.yyy:* formuliert;
- in einer expliziten Regel fehlt der Doppelpunkt oder vor dem Doppelpunkt ist kein Dateiname angegeben;
- in einer Makrodefinition fehlt das Gleichheitszeichen oder vor dem Gleichheitszeichen ist kein Makroname angegeben.

Division by zero
(Division durch Null)
Die Anwendung der Operatoren »Division« oder »Modulo« innerhalb einer *!if*-Direktive ergibt eine Division durch Null.

Expression syntax error in !if statement
(Ungültiger Ausdruck zusammen mit *!if*)
Mögliche Ursachen sind fehlende/überzählige Klammern, Operatoren usw.

File name too long
(MS-DOS-Dateiname mit mehr als 64 Zeichen)

Illegal character constant in expression X
(Nicht erlaubte Zeichenkonstante in Ausdruck X)
Wenn es sich dabei um ein Buchstaben handelt, dann haben Sie sich vermutlich bei der Angabe eines Makronamens vertippt.

Illegal octal digit
(»8« oder »9« in einer oktalen Konstanten)

Macro expansion too long
(Makro-Erweiterung erzeugt mehr als 4096 Zeichen)
Makros können nicht rekursiv formuliert werden, weil eine Abbruchbedingung fehlt: das Makro erweitert sich solange selbst, bis die zulässige Größe überschritten wird.

Misplaced elif statement
(*!elif*-Direktive an dieser Stelle nicht erlaubt)

Misplaced else statement
(*!else*-Direktive an dieser Stelle nicht erlaubt)

Misplaced endif statement
(*!endif*-Direktive an dieser Stelle nicht erlaubt)
In allen drei Fällen fehlt ein einleitendes *!if* bzw. MAKE hält den damit eingeleiteten Block aus irgendeinem Grund bereits für abgeschlossen.

No file name ending
(Dateiname in *!include* nicht mit " oder >abgeschlossen)

Redefinition of target 'XXXXXXXX'
(Datei 'XXXXXXXX' erscheint in mehr als einer expliziten Regel als *Ziel*)

Unable to open include file 'XXXXXXXX.XXX'
(Include-Datei 'XXXXXXXX.XXX' kann nicht geöffnet werden)
Diese Fehlermeldung erhalten Sie auch dann, wenn MAKE eine Schleifenbildung von Include-Dateien erkennt und daraufhin abbricht.

Unexpected end of file in conditional started on line #
(*!if*-Block vor Dateiende nicht abschlossen. Start der Bedingung mit Zeile Nummer *x*)
Mit *!if* und *!endif* gebildete Gruppen müssen innerhalb derselben Datei beginnen und enden. Diese Meldung erhalten Sie auch dann, wenn MAKE Ihre *!endif*-Angabe aus irgendeinem Grund zurückgewiesen hat.

Unknown preprocessor statement
(Unbekannte Direktive)
MAKE hat ein *!* als erstes Zeichen einer Zeile gefunden, darauf folgt etwas anderes als *include, if, elif, else, endif* oder *error*.

THELP – Syntax-Hilfe für alle ASCII-Editoren

THELP.COM ist ein speicherresidentes Programm, das die Syntax-Hilfestellung von Turbo Pascal (und Turbo C!) auch außerhalb der integrierten Entwicklungsumgebungen verfügbar macht – z.B. in einem eigenen Editor oder bei der Fehlersuche mit dem *Turbo Debugger*. THELP belegt nur 20 KByte im Hauptspeicher. Wenn THELP einmal installiert ist, reicht ein Druck auf die Taste 5 im numerischen Ziffernblock, um die Syntax-Hilfe von Turbo Pascal zu aktivieren, die eine Erläuterung zu dem Pascal-Schlüsselwort liefert, auf dem sich der Cursor gerade im Bildschirm befindet.

Hinweis: THELP ist über diverse Kommandozeilen-Parameter fast unendlich konfigurierbar – auch der Tastenbefehl zur Aktivierung läßt sich den eigenen Bedürfnissen anpassen (was speziell dann notwendig sein dürfte, wenn Ihr eigener Editor die Taste 5 selbst mit einer Funktion belegt).

Der Start von THELP

erfolgt von der DOS-Kommandoebene aus und hat die folgende Syntax:

```
THELP [Schalter/Parameter]
```

Auf die möglichen Schalter und Parameter kommen wir in einem der folgenden Abschnitte noch zurück – zuerst einmal sind nämlich einige »Vorsichts« am Platze:

Vorsicht: THELP ist ein *speicherresidentes* Programm – andere Programme wie Turbo Pascal, Turbo C usw. müssen zu diesem Zeitpunkt vollständig beendet sein. Wenn Sie die DOS-Kommandoebene via *File/DOS shell* (oder ähnliche

Befehle anderer Programme) aufrufen und dann THELP starten, dürften sich über kurz oder lang unerfreuliche Effekte ergeben. Falls Sie SideKick 1.x oder SideKick Plus verwenden: THELP muß *vor* diesen Programmen in den Speicher geladen werden.

Hinweis: Ein Aufruf ohne weitere Schalter und Parameter setzt voraus, daß sich die Datei mit den Texten der Hilfestellung (für Turbo Pascal: TURBO.HLP, für Turbo C: TCHELP.TCH) im selben Verzeichnis wie THELP.COM selbst befindet. Für dieses Verzeichnis ist THELP bei der Installation von Turbo Pascal mit INSTALL konfiguriert worden (was sich natürlich jederzeit ändern läßt – siehe */F* und */W* im Abschnitt »Kommandozeilen-Parameter«).

... und die Aktivierung

Nach einer Standard-Installation aktivieren Sie von THELP mit einem Druck auf die Taste 5 des numerischen Ziffernblocks (Scancode 04cH, ohne zusätzliche Modifizierer wie SHIFT, ALT o.ä). Diese Voreinstellung läßt sich über einen Kommandozeilen-Parameter beliebig ändern.

Nach der Aktivierung stehen zum einen dieselben Befehle wie innerhalb der integrierten Entwicklungsumgebung zur Verfügung – zum anderen kennt THELP noch eine Reihe weiterer Funktionen. Beides ist in Tabelle 10.1 zusammegefaßt:

Tabelle 10.1 Tastenbefehle nach der Aktivierung von THELP

Taste	Wirkung
← → ↑ ↓	bewegen den Cursor zwischen den einzelnen Wahlpunkten innerhalb des momentan gezeigten Hilfe-Bildschirms
PgUp , PgDn	blättern zwischen den einzelnen Seiten längerer Hilfstexte um
Shift ← → ↑ ↓	bewegen den Cursor, wenn ein Block markiert werden soll
Tab	bewegt den Cursor zum nächsten Schlüsselwort
Shift Tab	bewegt den Cursor zum vorherigen Schlüsselwort
Enter	bringt den Text zu dem Stichpunkt auf den Bildschirm, auf dem der Cursor innerhalb des Hilfstextes momentan steht.
Esc	beendet die Hilfestellung und bringt Sie zu dem Programm zurück, das durch die Aktivierung von THELP unterbrochen wurde.
F1	aktiviert das Inhaltsverzeichnis der Hilfestellung.
Alt F1	blättert rückwärts durch die Bildschirme der Hilfestellung, die zuvor aktiviert worden waren. THELP speichert die jeweils 20 letzten Bildschirme.
Ctrl F1	bringt eine Zusammenfassung der Tastenbefehle von THELP auf den Bildschirm.

5	Eine weitere Eingabe das Aktivierungsbefehls (normalerweise die Taste 5 auf dem numerischen Ziffernblock) hat nur auf Systemen mit zwei Monitoren eine Wirkung: In diesem Fall beendet THELP die Hilfestellung, läßt aber den zuletzt gezeigten Text auf dem inaktiven Monitor stehen.
Ctrl P	kopiert den Beispieltext in die Anwendung.

Entfernen von THELP

Der Befehl

```
THELP /U
```

entfernt THELP wieder aus dem Speicher (wobei allerdings vorausgesetzt wird, daß THELP das momentan »oberste« residente Programm ist, d.h. als letztes speicherresidentes Programm gestartet wurde).

Einfügen von Texten der Hilfestellung ins laufende Programm

Einzelne Schlüsselwörter und komplette Bildschirme der Hilfestellung lassen sich in das Programm einfügen, das durch die Aktivierung von THELP unterbrochen wurde. THELP simuliert dabei eine Eingabe über die Tastatur: falls Ihr Editor dabei Zeichen »verschluckt«, müssen Sie mit der Einfüge-Geschwindigkeit experimentieren, die sich über einen Kommandozeilen-Parameter verändern läßt.

Hinweis: Bei der Arbeit mit dem Editor der integrierten Entwicklungsumgebung müssen Sie entweder die Einfüge-Geschwindigkeit herabsetzen oder die automatische Einrückfunktion des Editors (mit Ctrl O I) abschalten.

Kommandozeilen-Parameter

Beim Start mit dem Parameter /H, /? oder ? gibt THELP den folgenden Text aus:

```
  USAGE:  THELP [options]
Command line options (must be separated by spaces):
  /Fname    Full path and filename of help file
  /H,/?,?   Display this help screen
  /Kxxyy    Change hotkey: xx=shift state, yy=scan code
  /Px       Pasting speed; 0=slow, 1=medium, 2=fast
  /S+-      Snow checking for video switch (default=S-)
```

```
  /U        Remove THELP from memory
  /W        Write Options to THELP.COM and exit
```

Frei übersetzt:

```
Syntax:   THELP [Schalter/Parameter]
Parameter werden durch einen Schrägstrich (/) eingeleitet und müssen
durch Leerzeichen voneinander getrennt werden; Angaben zu Parametern
müssen direkt (d.h. ohne Leerzeichen dazwischen) auf den Parameter-
namen folgen. Beispiel:
    THELP /Fd:\tp\hs\help\turbodt.hlp /W

Die folgenden Parameter sind definiert:
  /Fname    Suchweg und Name der Datei mit den Hilfstexten
  /H,/?,?   Ausgabe dieser Hilfestellung
  /Kxxyy    Aktivierungsbefehl ändern: xx=Modifizierer, yy=Scancode
  /Px       Einfüge-Geschwindigkeit: 0=niedrig, 1=mittel, 2=hoch
  /S+-      "Schnee" auf dem Bildschirm überprüfen
  /U        residente Version von THELP aus dem Speicher entfernen
  /W        Parameter in THELP.COM schreiben, keine Installation
            im Speicher
```

Details zu den Kommandozeilen-Parametern

/F*Name* – Suchweg/Dateiname der Hilfstexte

Normalerweise sucht THELP bei Start das momentan gesetzte Verzeichnis nach den Dateien TURBO.HLP (für Turbo Pascal) bzw. TCHELP.TCH (für Turbo C) ab – und verabschiedet sich mit einer Fehlermeldung, wenn keine dieser beiden Dateien dort gespeichert ist.

Der Parameter /F legt direkt fest, in welcher Datei THELP nach Texten der Hilfestellung suchen soll. Beispiele:

```
THELP /FC:\TP\TURBO.HLP
THELP /FC:\TURBOC\TCHELP.TCH
```

/H, /?, ? – Hilfestellung

Beim Aufruf mit einem dieser Parameter gibt THELP die im vorletzten Abschnitt beschriebene Zusammenfassung der Kommandozeilen-Parameter auf den Bildschirm aus und endet dann ohne Installation im Hauptspeicher.

/K*xxyy* – Aktivierungsbefehl ändern

Mit diesem Parameter läßt sich die Taste (oder Tastenkombination) bestimmen, über die THELP aktiviert werden soll. Beide Angaben sind in hexadezimaler Notation:

1. und 2. Digit (Modifizierer)

Taste	Wert
[Shift_Rechts]	01h
[Shift_Links]	02h
[Control]	04h
[Alt]	08h

Hinweis: Mehrere Modifizierer werden über OR miteinander verbunden.

3. und 4. Digit (Scancodes)

A	1eh	N	31h	0	0bh	F1	3bh
B	30h	O	18h	1	02h	F2	3ch
C	2eh	P	19h	2	03h	F3	3dh
D	20h	Q	10h	3	04h	F4	3eh
E	12h	R	13h	4	05h	F5	3fh
F	21h	S	1fh	5	06h	F6	40h
G	22h	T	14h	6	07h	F7	41h
H	23h	U	16h	7	08h	F8	42h
I	17h	V	2fh	8	09h	F9	43h
J	24h	W	11h	9	0ah	F10	44h
K	25h	X	2dh			F11	57h
L	26h	Y	15h			F12	58h
M	32h	Z	2ch				

Hinweis: Die Funktionstasten [F11] und [F12] stehen nur auf erweiterten Tastaturen (MF2) zur Verfügung.

Mit dem folgenden Aufruf ließe sich beispielsweise festlegen, daß THELP über die Tastenkombination [Shift_links][Alt]+K aktiviert wird:

```
THELP /K0A2C
```

Achtung: THELP arbeitet – zumindest, wenn dieses Programm *nach* KEYBGR installiert wird – mit einem direkten Zugriff auf die Tastatur. Folglich gelten hier die »unübersetzten« Scancodes der amerikanischen Originalbelegung (Y und Z vertauscht etc).

/P*x* – Einfüge-Geschwindigkeit

Zum Einfügen von Schlüsselwörtern und ganzen Bildschirmen der Hilfestellung (Tastenbefehle I bzw. P) simuliert THELP Eingaben über die Tastatur (d.h. setzt den Tastaturpuffer des Systems).

Einige Editoren (inklusive dem Editor der integrierten Entwicklungsumgebung bei aktivierter automatischer Einrückung auf einem Standard-PC mit 4,77 MHz) können die von THELP simulierten Werte nicht schnell genug verarbeiten und »verschlucken« unter Umständen Zeichen.

Mit dem Parameter /P läßt sich die Einfüge-Geschwindigkeit in drei Stufen einstellen:

/P1 »hoch« – THELP fügt neue Zeichen so schnell wie möglich in den Tastaturpuffer ein. Diese Einstellung ist die Standardvorgabe.

/P2 »mittel« – pro »Clock Tick« werden 4 Zeichen eingefügt (also rund 75 Zeichen pro Sekunde).

/P3 »niedrig« – THELP prüft bei jedem »Clock Tick« des Systems, ob der Tastaturpuffer leer ist – wenn ja, wird ein einzelnes Zeichen eingefügt.

/S – Schnee auf dem Bildschirm unterdrücken

Einige ältere CGA-Bildschirme zeigen gelegentlich Schnee auf dem Bildschirm. Sie können dies unterdrücken, indem Sie THELP mit der Option /S+ starten. Diese Einstellung kann über den Parameter /W als Standard definiert werden (siehe unten). Überprüfen auf Schnee dauert einige Zeit, benutzen Sie daher den Parameter nur dann, wenn er wirklich notwendig ist. Mit /S- wird die Option wieder deaktiviert (die Standardvorgabe).

/U – Residente Version von THELP aus dem Speicher entfernen

Beim Aufruf mit diesem Parameter wird die residente Version von THELP aus dem Speicher entfernt. THELP muß zu diesem Zeitpunkt das »oberste« Programm im Speicher sein:

- falls Sie andere speicherresidente Programme (wie SideKick (Plus)) *nach* dem Start von THELP geladen haben, müssen Sie diese Programme *zuerst* wieder entfernen;
- die »De-Installation« von THELP muß von der DOS-Kommandoebene aus geschehen (d.h. nicht über einen Aufruf von DOS via *File/DOS shell*).

/W – Parameter in THELP.COM schreiben

Beim Aufruf mit */W* modifiziert sich THELP selbst: die restlichen angegebenen Parameter werden direkt in die Datei THELP.COM geschrieben und stehen für nachfolgende Starts des Programms als Standard zur Verfügung. Nach dieser Modifikation endet THELP ohne eine Installation im Speicher. Alle Optionen können verwendet werden.

TOUCH – »Erneuern« von Dateien

TOUCH verändert Datum und Uhrzeit angegebener Dateien und macht Quelltexte ohne weitere Veränderung »neuer« als ihre Object-Dateien. Sinnvoll ist eine solche Aktion z.B. , wenn Sie die Fehlersuche in einem Projekt beendet haben und sämtliche Dateien ohne Prüfung von Indexen, des Stacks, zusätzlichen Debug-Informationen usw. neu compilieren lassen wollen.

TOUCH wird von der DOS-Kommandoebene aus mit der folgenden Syntax aufgerufen:

```
touch Dateiname [Dateiname ...]
```

Die Jokerzeichen * und ? sind hier erlaubt – ein Aufruf wie

```
touch *.pas \tp\bin\*.asm
```

bringt beispielsweise sämtliche Pascal-Quelltexte des momentan gesetzten Verzeichnisses sowie sämtliche Assembler-Quelltexte im Verzeichnis \TP\BIN des aktuellen Laufwerks auf das neueste Datum.

TPUMOVER – Die Bibliotheksverwaltung von Turbo Pascal

Wie in Kapitel 3 »Units, Overlays und andere Geheimnisse« erläutert, prüft Turbo Pascal bei der Suche nach einer Unit zuerst die Datei TURBO.TPL: dort enthaltene Units – wie z.B. *System*, *Overlay*, *Crt*... – werden beim Start beider Compiler automatisch in den Hauptspeicher geladen und stehen deshalb ohne weitere Diskettenzugriffe zur Verfügung.

Wenn Sie eine eigene Unit definiert (und gründlich ausgetestet) haben, die Sie häufig innerhalb Ihrer Programme verwenden, dann können Sie auch für diese Unit einiges an Zeit (für Datums- und Uhrzeitvergleiche, Ladeoperationen usw.) einsparen, indem Sie sie den vordefinierten Units von TURBO hinzufügen – mit dem Programm TPUMOVER ist das kein Problem.

Umgekehrt können Sie TPUMOVER natürlich auch verwenden, um selten benötigte Units aus TURBO.TPL zu entfernen – dadurch reduzieren Sie die Größe der Datei und gewinnen zusätzlichen Speicherplatz.

Einige technische Details

Die meisten Punkte, die im Zusammenhang mit Unit-Dateien zu berücksichtigen sind, haben wir in Kapitel 3 »Units, Overlays und andere Geheimnisse« bereits erwähnt. Der Übersichtlichkeit halber stellen wir sie hier noch einmal zusammen:

- Turbo Pascal unterscheidet zwei Arten von Unit-Dateien: Die Compilierung eines einzelnen Unit-Quelltextes erzeugt eine .TPU-Datei (Turbo Pascal Unit), die Zusammenfassung mehrerer .TPU-Dateien mit TPUMOVER erzeugt eine .TPL-Datei (Turbo Pascal Library).

- Fünf der acht Standard-Units von Turbo Pascal sind in der Datei TURBO.TPL zusammengefaßt, die beim Start der integrierten Entwicklungsumgebung bzw. der Kommandozeilen-Version automatisch geladen wird. TURBO.TPL ist gegenwärtig die einzige Bibliothek, die Turbo Pascal unterstützt.

- Turbo Pascal sucht eine über **uses** angegebene Unit zuerst in TURBO.TPL. Wenn die gesuchte Unit nicht Teil von TURBO.TPL ist, dann wird nach einer .TPU-Datei gesucht, die denselben Namen wie die Unit hat. Eine mit **uses** angegebene Unit *IntLib* hat also die Suche nach der Datei INTLIB.TPU zur Folge.

■ Die Compilierung eines Programms über *C/Compile* setzt voraus, daß alle mit **uses** angegebenen Units in compilierter Form vorliegen. Eine Compilierung über *C/Build* bzw. den Kommandozeilen-Parameter */B* sucht für alle Units, die nicht Teil von TURBO.TPL sind, zuerst die Quelltext-Dateien und übersetzt diese Units neu, bevor das Programm selbst compiliert wird. Die Compilierung eines Programms über *C/Make* (bzw. */M*) arbeitet nach demselben Prinzip, führt eine Recompilierung aber nur dann aus, wenn der Quelltext neueren Datums als die entsprechende .TPU-Datei ist.

■ Der Zugriff auf TURBO.TPL beim Start von Turbo Pascal erfolgt über den als *Turbo directory* gesetzten Suchweg, Zugriffe auf .TPU-Dateien dagegen über die als *Unit directories* angegebenen Suchwege.

Die Verwendung von TPUMOVER

Analog zur Kommandozeilen-Version des Compilers läßt sich auch TPUMOVER mit Parametern aufrufen. Die Syntax ist:

```
tpumover turbo /Parameter Unit-Dateiname
```

Mögliche *Parameter* und ihre Wirkung

+	fügt die durch *Unit-Dateiname* angegebene Unit zu TURBO.TPL hinzu
-	löscht die durch *Unit-Dateiname* angegebene Unit aus TURBO.TPL
*	kopiert die durch *Unit-Dateiname* angegebene Unit aus TURBO.TPL in die entsprechende Datei
?	gibt eine kurze Liste der verfügbaren Parameter aus

Einige Beispiele:

```
tpumover turbo /+ intlib   <- liest die Datei INTLIB.TPU und fügt die
                              Unit INTLIB zu TURBO.TPL hinzu
tpumover turbo /- intlib   <- löscht die Unit IntLib aus TURBO.TPL
tpumover turbo /* intlib   <- sucht die Unit IntLib in TURBO.TPL,
                              erzeugt die Datei INTLIB.TPU und
                              kopiert IntLib in diese Datei hinein
```

A. Unterschiede der Versionen

Dieser Anhang ist ausschließlich für Programmierer gedacht, die bereits mit früheren Versionen von Turbo Pascal gearbeitet haben. Da eine vollständige Liste der Neuerungen wenig sinnvoll erscheint (weil sie – zumindest für den Vergleich zwischen den Versionen 3.0 und 6.0 – die restlichen rund 800 Seiten der beiden Handbücher ausmacht), beschränken wir uns in beiden Fällen auf die konzeptuellen und programmiertechnischen Änderungen, die an existierenden Programmen *notwendig* sind.

Compilierung von Programmen der Version 4.0

Wie gesagt – in diesem Abschnitt geht es *nicht* um Overlays, den Debugger, prozedurale Variablen usw., sondern lediglich um die Punkte, die bei der Übersetzung eines Programms zu beachten sind, das für die Version 4.0 des Compilers geschrieben wurde. Trotz der umfangreichen Erweiterungen seit der Version 5.0 ist die Liste erfreulich kurz.

Allgemeines

- Die integrierte Entwicklungsumgebung gibt nach dem Ende eines Programms nicht länger die Meldung `Press any key...` aus, sondern schaltet sofort wieder auf den »Turbo-Bildschirm« um; die Ausgaben des Programms lassen sich über den Tastenbefehl Alt F5 sichtbar machen.

- Der Editor verfügt neben anderen Erweiterungen über eine Füll-Funktion, die aufeinanderfolgende Leerzeichen automatisch in eine möglichst platzsparende Kombination von Leerzeichen und Tabs verwandelt. Wenn Sie ein eigenes Programm zur Ausgabe und/oder Übernahme von Listings verwenden, sollten Sie dieses Programm entweder entsprechend modifizieren – oder die Füll-Funktion mit Ctrl OF abschalten.

- Die integrierte Entwicklungsumgebung prüft beim Start, ob der Computer mit einer EMS-Karte ausgerüstet ist – wenn ja, werden bis zu 64 KByte auf dieser Karte für den Puffer des Editors belegt, um Platz im Hauptspeicher zu sparen.

(Bei eventuellen Konflikten läßt sich diese Option über das Installationsprogramm TINST abschalten).

■ In der Version 6.0 setzt die Lokalisierung von Laufzeitfehlern innerhalb der integrierten Entwicklungsumgebung nicht einen, sondern zwei Schalter im Zustand *gesetzt* voraus: *O/C/Debug information* (wie in der Version 4.0) und *O/Debugger/Integrated*.

.TPM und .MAP-Dateien

Ab der Version 5.0 erzeugt der Linker keine .TPM-, sondern .MAP-Dateien; dementsprechend wurde der Compiler-Befehl {*$T*} durch Menüpunkte bzw. Kommandozeilen-Parameter ersetzt (siehe folgende Abschnitte).

Nullstrings und die Funktion Length

Seit der Version 5.0 codiert der Compiler Vergleiche und Zuweisungen mit Nullstrings, also

```
StringVar := '';
if StringVar = '' then...
```

als direkten Zugriff auf *StringVar*[*0*] (d.h. das Längenbyte). Die in der Version 4.0 von einigen Programmieren verwendeten manuellen Optimierungen wie

```
StringVar[0] := #0;
Length(StringVar) := 0;

if Length(StringVar) = 0 then ...
```

sind damit unnötig geworden. Mehr noch: die Verwendung eines Funktionsaufrufs auf der linken Seite einer Zuweisung erzeugt ab der Version 5.0 nun – korrekterweise – eine Fehlermeldung des Compilers.

Gleitkommazahlen

Die Verwendung der IEEE-Typen *Single..Extended* setzt ab der Version 5.0 nicht mehr voraus, daß das System mit einem Coprozessor ausgerüstet ist; aus Gründen der Kompatibilität wurde der Schalter {*$N*} aber auf der Standardvorgabe {*$N-*} (=*Real*-Variablen) belassen.

Die Übergabe von IEEE-Realtypen als Wertparameter erfolgt seit der Version 5.0 nicht mehr über die Register des Coprozessors, sondern (analog zu allen anderen Werten) durch den Stack des 8086. Assembler-Routinen, die sich auf gesetzte Register des Coprozessors verlassen, müssen um entsprechende Ladebefehle erweitert werden.

Units

- .TPU-Dateien (d.h. compilierte Units) beider Versionen sind nicht kompatibel, sämtliche Unit-Quelltexte müssen also mit der Version 6.0 einmal recompiliert werden.

- In der Version 6.0 sind die »Kompatibilitäts-Units« *Turbo3* und *Graph3* als eigenständige Dateien (im Unterverzeichnis TURBO3) gespeichert, d.h. nicht mehr Bestandteil von TURBO.TPL (was sich natürlich mit TPUMOVER jederzeit wieder ändern läßt).

- Die Version 4.0 beschränkt sich beim Vergleich von Deklarationen zwischen Interface-Teil und Implementationsteil einer Unit auf den Bezeichner und (bei Funktionen) den Ergebnistyp – Angaben wie

```
interface
  function Myfunc(X,Y: Integer): Integer;
implementation
  function Myfunc: Integer;
```

werden vom Compiler ohne Beanstandung übersetzt. Seit der Version 5.0 wurde diese Prüfung in zwei Richtungen erweitert und erlaubt nun *entweder* eine reine Namensangabe im Interface zusammen mit einer vollständigen Deklaration in der Implementation, also

```
interface
  function Myfunc;
implementation
  function Myfunc(X,Y: Integer): Integer;
```

oder es wird eine völlige Übereinstimmung zwischen beiden Teilen gefordert:

```
interface
  function Myfunc(X,Y: Integer): Integer;
implementation
  function Myfunc(X,Y: Integer): Integer;
```

Die zweite Form erfordert etwas mehr Tipparbeit (wenn man nicht sowieso die Kopierfunktion des Editors benutzt), bietet aber einen wesentlich besseren Schutz gegen eventuelle Unstimmigkeiten.

Compiler-Befehle im Quelltext

Die folgenden Compiler-Befehle im Quelltext haben in der Version 6.0 eine andere Funktion bzw. werden nicht mehr unterstützt:

$U – Unit-Dateiname

In der Version 6.0 ergibt sich der Name einer Unit grundsätzlich aus dem Namen des Quelltextes. Aus einem Quelltext MYUNIT.PAS erzeugt der Compiler also immer eine Datei namens MYUNIT.TPU – und reagiert auf den nicht mehr

definierten Befehl {*$U Unitname*} mit einer Fehlermeldung. Das Verzeichnis, in dem .TPU-Dateien gespeichert werden, läßt sich über *O/D/EXE&TPU directory* bzw. den Kommandozeilen-Parameter */E<Suchweg>* festlegen; die Angabe der Suchwege für Unit-Quelltexte erfolgt über *O/D/Unit directories* bzw. den Kommandozeilen-Parameter */U<Suchweg>*.

$T – TPM-Datei

Der Linker der Version 6.0 erzeugt keine .TPM-, sondern .MAP-Dateien (die nur für die Fehlersuche mit einem eigenständigen Debugger benötigt werden) – deshalb reagiert der Compiler auf den Befehl {*$T+*} bzw. {*$T-*} im Quelltext mit einer Fehlermeldung. Wie detailliert die Angaben in einer .MAP-Datei ausfallen, läßt sich über das neue Untermenü *Options/Linker/Map file* der integrierten Entwicklungsumgebung bzw. die Parameter */GS*, */GP* und */GD* der Kommandozeilen-Version von Turbo Pascal festlegen.

$L – Link-Puffer

In der Version 6.0 legt dieser Schalter fest, ob der Compiler zusätzlich Informationen über lokale Symbole in der .EXE-Datei speichert bzw. ob diese Daten in eine vom Linker erzeugte .MAP-Datei aufgenommen werden.

Die Festlegung, ob der Linker den Hauptspeicher oder eine Zwischendatei zur Pufferung benutzt, erfolgt über den Menüpunkt *O/L/Link buffer* der integrierten Entwicklungsumgebung bzw. den Kommandozeilen-Parameter */L*.

$R – Bereichsüberprüfung

In der Version 6.0 ist die Standardvorgabe dieses Schalters *Off*; Funktion und Arbeitsweise sind gleich geblieben.

Lokale und globale Variablen

Die Anordnung lokaler Variablen auf dem Stack geschieht seit der Version 5.0 in anderer Reihenfolge als in der Version 4.0 (was die »Abwärtskompatibilität« allerdings nur beeinflußt, wenn bei **inline**-Routinen mit direkter Adressierung über BP anstelle von Variablen- und Parameternamen gearbeitet wird).

Die Versionen ab 5.0 arbeiten standardmäßig mit wortweiser Ausrichtung aller Variablen, die mehr als ein Byte Speicherplatz belegen. »4.0-Programme«, die sich auf eine bestimmte *Abfolge* von Variablen verlassen, müssen diese Optimierung mit dem (neuen) Compiler-Befehl {*$A-*} unterdrücken.

Der Aufruf von TPC.EXE

wurde analog zu den Compiler-Befehlen im Quelltext geändert:

- Der Parameter */T* (TPM-Datei) wird durch einen der drei Parameter */GS*, */GP*

oder */GD* für .MAP-Dateien ersetzt (siehe Kapitel 9, »Kommandozeilen-Version«).

■ Der Parameter */$L* (Link-Puffer) hat seit der Version 5.0 den Namen */L* und eine etwas abweichende Syntax: Nur wenn */L* angegeben ist, verwendet der Linker eine Diskettendatei, ansonsten wird der Hauptspeicher benutzt.

Dazu kommen zwei weitere Änderungen: die Parameter */R* (Compilierung in den Hauptspeicher und Start) sowie */X* (Compilierung als Datei und Start) werden ab der Version 5.0 nicht mehr unterstützt. (*/X* ließe sich allerdings recht einfach durch eine entsprechende Batch-Datei nachkonstruieren).

Compilierung der Toolboxen

Kompatibilitätsprobleme mit dem Quelltext des Turbo Pascal Tutor und der Toolboxen gibt es keine. Lediglich einige Compiler-Schalter im Quelltext müssen neu gesetzt werden:

Database Toolbox
TAINST.PAS und TABUILD.PAS: Stellen Sie diesen Quelltexten den Befehl {*$A-*} voran.

Editor Toolbox

EDDIRECT.INC & MSDIRECT.INC: Stellen Sie diesen den Quelltexten den Befehl {*$A-*} voran, löschen Sie den Befehl {*$T+*} und fügen Sie den Befehl {*$L+*} nach der Zeile mit dem Befehl {*$D+*} ein.

EDINST.PAS, MSINST.PAS, & INSTALL.PAS: Stellen Sie der Zeile mit dem Befehl `{$V-}`den Befehl `{$A-}`voran.

BINED.PAS: Stellen Sie der Zeile mit dem Befehl `{$I-}`den Befehl `{$A-}`voran.

Achtung: Sämtliche .TPU-Dateien der Toolboxen müssen vor ihrer Verwendung mit der Version 6.0 recompiliert werden!

Modifizierte Prozeduren und Funktionen

Seit der Version 5.0 wurden lediglich vier Routinen gegenüber der Version 4.0 modifiziert:

New wurde ab der Version 5.5 erweitert, damit man den Aufruf eines Konstruktors als zweiten Parameter zum Anlegen einer dynamischen Objektvariablen optional angeben kann.

Dispose wurde ab der Version 5.5 erweitert, damit man den Aufruf eines Destruktors als zweiten Parameter zum Freigeben einer dynamischen Objektvariablen optional angeben kann.

ParamStr liefert beim Betrieb unter DOS 3.x beim Aufruf mit dem Wert 0 als Argument keinen Nullstring, sondern Suchweg und Namen des Programms.

■ *SetUserCharSize* erfordert in der Version 4.0 einen Aufruf von *SetTextStyle*, damit der gesetzte Vergrößerungsfaktor wirksam wird. Ab der Version 5.0 ist dieser Aufruf nicht notwendig (tatsächlich muß er gestrichen werden – sonst setzt *SetTextStyle* wieder eine eigene Zeichengröße).

Erweiterungen der Laufzeitbibliothek

Die Laufzeitbibliothek wurde seit der Version 5.0 um rund zwei Dutzend Prozeduren und Funktionen erweitert (wodurch sich natürlich keine Probleme bei der Übernahme existierender Programme ergeben – die folgende Tabelle soll Ihnen lediglich die Suche abnehmen):

Name	Unit
DosVersion-Funktion	Dos
EnvCount-Funktion	Dos
EnvStr-Funktion	Dos
FExpand-Funktion	Dos
FillEllipse-Prozedur	Graph
FSearch-Funktion	Dos
FSplit-Prozedur	Dos
GetCBreak-Prozedur	Dos
GetDefaultPalette-Funktion	Graph
GetDriverName-Funktion	Graph
GetEnv-Funktion	Dos
GetMaxMode-Funktion	Graph
GetModeName-Funktion	Graph
GetPaletteSize-Funktion	Graph
GetVerify-Prozedur	Dos
InstallUserDriver-Funktion	Graph
InstallUserFont-Funktion	Graph
OvrClearBuf-Prozedur	Overlay
OvrGetBuf-Funktion	Overlay
OvrGetRetry-Funktion	Overlay
OvrInit-Prozedur	Overlay
OvrInitEMS-Prozedur	Overlay
OvrSetBuf-Prozedur	Overlay
OvrSetRetry-Prozedur	Overlay
RunError-Prozedur	System
Sector-Prozedur	Graph
SetAspectRatio-Prozedur	Graph
SetCBreak-Prozedur	Dos
SetRGBPalette-Prozedur	Graph

Name	*Unit*
SetVerify-Prozedur	Dos
SetWriteMode-Prozedur	Graph
SwapVectors-Prozedur	Dos

Eine ausführliche Beschreibung dieser neuen Routinen finden Sie in den entsprechenden Abschnitten des Kapitels 1, »Verzeichnis der Routinen«, des Referenzhandbuches.

Syntax- und Laufzeitfehler

Die folgenden Listen sollen Ihnen lediglich das Nachschlagen erleichtern – einen Einfluß auf die Compilierung oder Ausführung existierender Programme haben die Veränderungen nicht.

Nicht mehr existierende Fehlercodes des Compilers

Die folgenden Fehlermeldungen und -codes der Versionen 4.0, 5.0 und 5.5 existieren in der Version 6.0 nicht mehr bzw. wurden durch andere Meldungen ersetzt:

24 »File componets may not by files« – ab der Version 5.5 prüft der Compiler die Komponenten zusätzlich auch daraufhin, ob es sich um Objekttypen handelt.

33 »Type identifier expected« – ab der Version 6.0 wird geprüft, ob der Bezeichner einen Zeiger-Typ repräsentiert.

99 »Files are not allowed here« – ab der Version 5.5 wird zusätzlich geprüft, ob Prozedur-Typen verwendet wurden.

108 »Not enough memory to run program« – ab der Version 5.0 prüft die integrierte Entwicklungsumgebung *vor* dem Start eines Programms, ob genug Platz im Hauptspeicher vorhanden ist und reagiert gegebenenfalls entsprechend.

109 »Cannot find .EXE file« – in der Version 6.0 ist die integrierte Entwicklungsumgebung recht unempfindlich gegenüber gewechselten Verzeichnissen und plötzlich verschwundenen .EXE-Dateien – nötigenfalls wird vor den Start des Programms ein weiterer Aufruf des Compilers gesetzt.

115 »Must have an 8087 to compile this« – die Version 6.0 enthält einen Emulator, der IEEE-Gleitkommazahlen auch auf Computern ohne Coprozessor bearbeiten kann.

119 »TPM file format error« – der Linker der Version 6.0 erzeugt keine .TPM-, sondern .MAP-Dateien, die nur für die Fehlersuche mit eigenständigen Debuggern notwendig sind – die integrierte Entwicklungsumgebung kommt ohne sie aus.

125 »Module has no debug information« – in der Version 6.0 prüft die integrierte Entwicklungsumgebung vor einer Suche via *Find Error*, ob das Programm Informationen zur Fehlersuche enthält – falls nicht, bekommen Sie nur das »Fehlerfenster« selbst zu sehen (und keine zweite Meldung, daß sich der Fehler nicht lokalisieren läßt).

133 »Old TPM file« – ist wie der Code 119 überflüssig geworden und wurde redefiniert (siehe nächster Abschnitt).

Eine genaue Beschreibung aller Fehlercodes des Compilers der Version 6.0 finden Sie in Kapitel 3 »Fehlermeldungen und -codes« des *Referenzhandbuches*.

Laufzeitfehler

Die Version 6.0 definiert gegenüber der Version 4.0 lediglich sieben neue Laufzeitfehler, die im Zusammenhang mit Overlays, Objekten und Turbo Vision stehen:

208 Overlay manager not installed (ab Version 5.0)
209 Overlay file read error (ab Version 5.0)
210 Object not initialized (ab Version 5.5)
211 Call to abstract method (ab Version 6.0)
212 Stream registration error (ab Version 6.0
213 Collection index out of range (ab Version 6.0)
214 Collection overflow errror (ab Version 6.0)

Auch hier verweisen wir für die entsprechenden Beschreibungen auf Kapitel 3,»Fehlermeldungen und -codes« des Referenzhandbuches.

Konvertierung von Programmen der Version 3.0

Obwohl die Version 6.0 von Turbo Pascal viele Erweiterungen gegenüber der Version 3.0 und eine ganze Reihe neuer Konzepte enthält, sind beide in hohem Maße »kompatibel« – außerdem haben wir mehrere Hilfen zur Übernahme existierender Programme zur Verfügung gestellt:

- Die Unit *Turbo3* deklariert mehrere Standard-Bezeichner, die ab der Version 4.0 nicht mehr direkt unterstützt werden.

- Die Unit *Graph3* enthält die Grafik-Routinen der Version 3.0 inklusive der Turtlegrafik.

Die Aufnahme dieser beiden »Kompatibilitäts-Units« und die damit verbundenen Möglichkeiten sind in Kapitel 12, »Standard-Units«, des Programmierhandbuches besprochen. Wir konzentrieren uns erst einmal auf die Unterschiede in der Arbeit mit dem Compiler und gehen dann auf technische Details ein:

Die Arbeit mit Turbo Pascal

Integrierte Entwicklungsumgebung vs. Kommandozeilen-Version

Die Version 6.0 bietet zwei Versionen des Compilers, die sich nur in ihrer Bedienung unterscheiden – intern arbeiten beide auf dieselbe Weise:

- die integrierte Entwicklungsumgebung (TURBO.EXE) faßt Compiler und Editor in einem System zusammen, das (wie Turbo C++) menü- und fensterorientiert arbeitet, mit einer ständig verfügbaren Hilfestellung versehen ist und eine automatische Fehlersuche durchführt.

  Schalter des Compilers werden über Menüpunkte gesetzt (soweit sie nicht als Befehle im Quelltext angegeben sind). Die neue IDE unterstüzt nun auch die Bedienung mit einer Maus.

- die Kommandozeilen-Version des Compilers (TPC.EXE) ist traditionellen Compilern ähnlicher und wird von der DOS-Kommandoebene aufgerufen. Schalter des Compilers werden über Kommandozeilen-Parameter gesetzt (soweit sie nicht im Quelltext angegeben sind).

Beide Versionen des Compilers haben exakt denselben Befehlsumfang, dieselben Möglichkeiten und können über MAKE.EXE aufgerufen werden, um mehrere Module eines Programms zu bearbeiten (siehe Kapitel 10 »Die Zusatzprogramme«). Die Kommandozeilen-Version belegt erheblich weniger Speicherplatz.

Bedingte Compilierung

Die Version 6.0 ermöglicht die Compilierung von Programmteilen in Abhängigkeit von definierten Symbolen. Um verschiedene Versionen eines Programms (beispielsweise für unterschiedliche Computermodelle) zu erstellen, ist bei entsprechender Formulierung des Quelltextes lediglich die Definition der jeweiligen Symbole beim Aufruf des Compilers erforderlich (siehe Kapitel 2, »Compiler-Befehle«, des Referenzhandbuches).

Separate Compilierung und Units

Die Version 6.0 erlaubt eine modulare Compilierung: Teile eines Programms können als *Units* einzeln getestet und übersetzt werden. Gegenüber dem (nach wie vor unterstützten) Konzept der Include-Dateien ergeben sich hier zwei Vorteile:

- erheblich kürzere Compilierungszeiten. Eine Unit wie *Graph* (mit mehreren tausend Zeilen Quelltext) wird nicht jedesmal neu compiliert, sondern nur noch in ein Programm aufgenommen;

- die Möglichkeit, komplette Funktionsgruppen als eigene Bibliotheken anzulegen, die nicht nur äußerst einfach in Programme aufgenommen, sondern auch ohne Preisgabe des Quellcodes weitergegeben werden können.

Compile, Make und Build

Der Menüpunkt *Compile* bzw. der Aufruf der Kommandozeilen-Version ohne spezielle Zusatzparameter ist praktisch mit dem Aufruf des Compilers in der Version 3.0 identisch: Turbo Pascal übersetzt den Quelltext, der sich momentan im Editor befindet bzw. die angegebene Datei, wobei Include-Dateien mit einbezogen und angegebene Units in compilierter Form aufgenommen werden.

Bei jeder modularen Programmiersprache ergibt sich das Problem der »Versionsalter« beteiligter Module: wenn Modul A auf Routinen des Moduls B zurückgreift, das seinerseits von Modul C abhängig ist, dann erfordert eine Änderung des Quelltextes von Modul A nicht nur die erneute Compilierung dieses Moduls, sondern auch der Module B und C. Turbo Pascal begegnet dieser Problematik mit der Funktion *Make*, die für jedes an einer Compilierung beteiligte Modul Datums und Uhrzeit von Quelltext und Object-Datei vergleicht: wenn der Quelltext neueren Datum als die Object-Datei ist, wird er automatisch neu übersetzt. Der Vergleich arbeitet über eine Baumstruktur und stellt so auch bei mehrfach verschachtelten Abhängigkeiten sicher, daß wirklich alle Teile eines Programms auf dem neuesten Stand sind.

Die Funktion *Build* arbeitet nach demselben Schema wie *Make*, nur daß hier kein Vergleich von Datum und Uhrzeit stattfindet – es werden grundsätzlich alle beteiligten Quelltexte neu übersetzt. *Build* kommt hauptsächlich dann zum Einsatz, wenn bei bedingter Compilierung Symbole geändert werden (nicht aber die Quelltexte selbst).

Hinweis: Eine Beschreibung der entsprechenden Parameter für die Kommandozeilen-Version (*/M* und */B*) finden Sie in Kapitel 9 »Kommandozeilen-Version«; eine eingehende Erläuterung des Programms MAKE.EXE, das noch wesentlich weitergehendere Möglichkeiten zur automatisierten Verwaltung von Projekten eröffnet, in Kapitel 10 »Die Zusatzprogramme«.

Es würde den Rahmen dieses Kapitels sprengen, alle Unterschiede und Vorzüge der neuen IDE hier zu erläutern. Wir verweisen daher auf die Kapitel 1 »Die neue integrierte Entwicklungsumgebung« und auf Kapitel 7 »Die IDE-Referenz«.

Die Implementation des Compilers

Obwohl die Version 6.0 des Turbo Pascal-Compilers eine Unzahl von Erweiterungen enthält und intern völlig anders arbeitet, können wir sie guten Gewissens als »kompatibel« zu ihren Vorläufern bezeichnen. Die folgenden Abschnitte beschreiben deshalb hauptsächliche eine Reihe kleinerer Änderungen, die sich durch die Erweiterung verschiedener Funktionen ergeben haben.

Soweit erforderlich, haben wir bei der Beschreibung der Unterschiede Vorschläge angefügt, wie Programme älteren Datums angepaßt werden können. Weitere Informationen finden Sie im zweiten Teil dieses Anhangs.

Deklarationen

In der Version 3.0 kann zusammen mit dem reservierten Wort **program** ein beliebiger Name verwendet werden, weil der Compiler diese Angabe sowieso völlig ignoriert. Die Version 6.0 verlangt, daß der Programmname einzigartig ist – er darf weder innerhalb des Programms noch innerhalb einer aufgenommenen Unit ein zweites Mal deklariert werden. Der Grund liegt darin, daß die Version 6.0 die Angabe »qualifizierter« Bezeichner erlaubt und Programmnamen hier eine echte Funktion haben.

Ein Beispiel: wenn in einem Programm die (im Unit *System* enthaltene) Standard-Funktion *Abs* neu deklariert wird, dann bezieht sich der Aufruf

```
x := Abs(-3.4);
```

auf die neu deklarierte Routine. Durch Voranstellen des Unitnamens wird ein qualifizierter Bezeichner gebildet – der folgende Aufruf aktiviert also die Funktion *Abs* der Unit *System*:

```
x := System.Abs(-3.4);
```

Details zu qualifizierten Bezeichnern finden Sie in Kapitel 3 »Units, Overlays und andere Geheimnisse« sowie in Kapitel 1 »Sprachelemente und Konstanten« *des Programmierhandbuches.* In der Version 3.0 müssen sämtliche Bezeichner (Konstanten, Variablen, Datentypen, Prozeduren und Funktionen) in einem Arbeitsgang compiliert werden und sich deshalb entweder im Quelltext der Haupt-Datei oder in einer via {*$I*} aufgenommenen Datei befinden. Die Version 6.0 erlaubt nicht nur die separate Compilierung von Prozeduren und Funktionen als Unit – Units können auch Deklarationen von Variablen, Konstanten und Datentypen enthalten, auf die innerhalb des Programms Bezug genommen wird. (Diese Erweiterung macht keine Änderungen existierender Programme erforderlich).

Die Version 3.0 begrenzt den Gesamtumfang des Programmcodes auf 64 KByte und erzeugt Dateien des Typs .COM, zur Realisierung größerer Programme müssen entweder Overlays oder Chain-Dateien verwendet werden. In der Version 6.0 ist der maximale Code-Umfang nicht einmal mehr durch die Größe des Hauptspeichers begrenzt; Programme werden hier als .EXE-Dateien gespeichert und können sich aus einer beliebigen Anzahl von Units mit *jeweils* maximal 64 KByte Code zusammensetzen; die Overlay-Verwaltung arbeitet wesentlich modularer, schneller – und dazu praktisch unsichtbar für den Programmierer.

Die Konvertierung von Programmen der Version 3.0, die mit *Chain* arbeiten, ist recht einfach: Entweder ersetzen Sie die Aufrufe von *Chain* und *Execute* durch Aufrufe der neu eingeführten Prozedur *Exec* (in der Unit *Dos*), oder Sie compilieren Ihre Chain-Dateien als Units und nehmen sie via **uses** direkt in das Hauptprogramm auf. Programme, die mit Overlays arbeiten, erfordern aufgrund des geänderten Konzepts grundsätzlich eine Unterteilung in Units (die dann *en bloc* als Overlays deklariert werden).

Compiler-Befehle

Beide Versionen des Compilers unterstützen Befehle innerhalb des Quelltextes, die als Kommentare »getarnt« sind und mit einem Dollarzeichen ($) eingeleitet werden. In der Version 6.0 sind einige neue Befehle hinzugekommen, die Bezeichnung anderer hat sich geändert. Tabelle A.1 gibt die Befehle der Version 6.0 wieder (eine ausführliche Beschreibung finden Sie in Kapitel 2 »Compiler-Befehle« des Referenzhandbuches):

Tabelle A.1 Compiler-Befehle der Version 6.0

Befehl	Standard	Beschreibung
`$A+/-`	+	Ausrichtung von Variablen im Speicher (- =ohne, + =wortweise)
`$B+/-`	-	boolsche Auswertung (- =Kurzschluß, + =komplett)
`$D+/-`	+	Debugger-Zusatzinformationen (+=ja, - =nein)
`$E+/-`	+	Emulation des Coprozessors, falls notwendig (+=ja, - =nein)
`$F+/-`	-	*far*-Aufrufe (- =nur wenn notwendig, + =immer *far*)
`$G+/-`	-	Erzeugung von Maschinencode für den 80286-Prozessor (+=ja, - =nein)
`$I+/-`	+	I/O-Prüfung (- =aus, + =an)
`$I <Name>`		Include-Datei aufnehmen
`$L+/-`	+	Debugger-Informationen über lokale Symbole
`$L <Name>`		Object-Datei aufnehmen
`$M s,l,h`		Speicherbelegung (Stack, MinHeap, MaxHeap). Vorgabe: 16384, 0, 655360
`$N+/-`	-	IEEE-Gleitkommatypen (- =nein (->*Real*), + =ja)
`$R+/-`	-	Bereichsüberprüfung (- =aus, + =an)
`$S+/-`	+	Stack-Prüfung (- =aus, + =an)
`$O <Name>`		Deklaration einer Unit als Overlay
`$V+/-`	+	**var**-Stringprüfung (+=an, - =aus)
`$X+/-`	+	erweiterte Syntax (+=an, - =aus)

Einige Hinweise dazu:

- Die Compiler-Befehle *$B*, *$D* und *$F* haben in der Version 6.0 dieselbe Syntax wie in der Version 3.0 – aber eine völlig andere Bedeutung (siehe Kapitel 2 »Compiler-Befehle« des Referenzhandbuches).

- Die Standardvorgabe für die Bereichsüberprüfung in der Version 6.0 ist *Off* – das Einfügen des Compiler-Befehls *{$R-}* zu Beginn des Quelltextes ist also nicht mehr notwendig, um in diesem Punkt dieselben Bedingungen wie bei der Version 3.0 herzustellen.

- Die Version 3.0 bewertet zusammengesetzte logische Ausdrücke in der Reihenfolge der Angabe, die Versionen seit 4.0 arbeiten dagegen *innerhalb* eines Ausdruckes von rechts nach links. Bei 3.0-Programmen, die sich auf diese

Auswertungsfolge verlassen, kann es deshalb trotz Unterdrückung des Kurzschlußverfahrens mit {$B+} zu Kompatibilitätsproblemen kommen. Ein Beispiel: Gegeben seien die Funktionen *Func1* und *Func2* sowie der Ausdruck

```
if Func1 and Func2 then ...
```

Während die Version 3.0 hier zuerst *Func1* und dann *Func2* aufruft, arbeiten die Versionen 4.0 und 6.0 genau in der umgekehrten Reihenfolge – falls diese beiden Funktionen mit Seiteneffekten arbeiten, muß der Ausdruck deshalb folgendermaßen umgestellt werden:

```
{$B+} if Func2 and Func1 then ...
```

Analoges gilt für Verkettungen mehrerer Ausdrücke:

```
if Func1 and Func2 and Func3 then ...
```

Das exakte Analogon der Versionen 4.0 und 6.0 ist in diesem Fall:

```
{$B+} if Func3 and Func2 and Func1 then ...
```

Eine vollständige Erklärung dieser Spezialfälle würde eine Analyse des lexikalischen Parsers von Turbo Pascal voraussetzen (was wir hier nicht vorhaben – uns ging es nur um einen Fingerzeig bei möglichen Problemen).

- Der Befehl {*$I <Dateiname>*} kann in der Version 3.0 an beliebiger Stelle stehen, die Datei selbst darf Deklarationen, komplette Prozeduren oder einfach einige Befehle enthalten. In der Version 6.0 darf der Befehl {*$I <Dateiname>*} nicht innerhalb eines Programmblocks (d.h. zwischen **begin** und **end**) gegeben werden. Wenn die Include-Datei ausführbare Pascal-Befehle enthält, erwartet der Compiler einen kompletten Programmblock innerhalb der Include-Datei, also vollständig abgeschlossene Prozeduren und/oder Funktionen oder das gesamte Hauptprogramm (inklusive **begin** und **end**).
  Zwischen *$I* und dem Dateinamen muß sich mindestens ein Leerzeichen befinden (das in der Version 3.0 optional ist).

- Die Verschachtelung von Include-Dateien ist in der Version 3.0 nicht möglich: Wenn eine Datei namens TEIL1.PAS mit {*$I TEIL1*} aufgenommen wird, meldet der Compiler einen Fehler, falls in TEIL1.PAS weitere Include-Befehle enthalten sind. Die Version 6.0 unterstützt Verschachtelungen von Include-Befehlen bis zu einer maximalen Tiefe von 15. (Diese Erweiterung macht keine Änderungen existierender Programme erforderlich).

- In der Version 3.0 können die Größe von Stack und Heap nur über Menüpunkte gesetzt werden. Die Version 6.0 unterstützt darüber hinaus einen Compiler-Befehl ({*$M*}) innerhalb des Programmtextes.

Eingaben über die Tastatur

Durch die Anpassung der Versionen 4.0 bis 6.0 an den ANSI-Standard ergibt sich hier ein recht subtiles Kompatibilitätsproblem: Die Version 3.0 behandelt

die Standardeingabe *Input* (d.h. die Tastatur) auf spezielle Weise – die neueren Versionen sehen dagegen keinen Unterschied zwischen *Input* und einer beliebigen anderen Textdatei. Als Folge dieser Spezialbehandlung werden *Read* und *Readln* in der Version 3.0 grundsätzlich mit einem Zeilenvorschub beendet (den *Readln* auch auf den Bildschirm ausgibt):

- Überzählige Eingabeelemente werden ignoriert – bei einem Aufruf wie *Readln(A,B)*; und der Eingabe *10 20 30* Enter ordnet *Readln* der Variablen *A* den Wert 10 zu, der Variablen *B* den Wert 20 – und »vergißt« die dritte Eingabe.

- Wenn eine Eingabe weniger Elemente enthält als angefordert, dann füllt die Version 3.0 den Eingabepuffer mit Ctrl-Z-Zeichen – bei einem Aufruf wie *Readln(A,B)*; und der Eingabe *10* Enter bekommt die Variable *A* den Wert 10 zugewiesen, *B* bleibt unverändert (solange es sich dabei um eine numerische Variable handelt – wenn *B* für einen String steht, ordnen *Read* und *Readln* hier einen Nullstring zu).

Seit der Version 4.0 wird die Tastatur wie eine normale Datei behandelt – und Enter damit wie ein normales Trennzeichen zwischen zwei Elementen:

- Überzählige Eingabeelemente bleiben »stehen«. Bei einem Aufruf wie *Readln(A,B)*; und der Eingabe *10 20 30* Enter liest das Programm die ersten beiden Werte, ordnet sie zu und beläßt die dritte Eingabe im Puffer – sie wird durch die nächste Eingabeanforderung als erstes Element gelesen.

- Wenn eine Eingabe weniger Elemente enthält als angefordert, dann wartet *Readln* einfach auf eine weitere »Zeile« – die beiden durch ein Aufruf wie *Readln(A,B)*; angeforderten Werte lassen sich also entweder als *10 20* Enter, als *10* Enter *20* Enter oder auch als *10* Enter Enter Enter ... *20* Enter eingeben.

Etwas technischer: *Read* überspringt in der Version 6.0 bei der Eingabe numerischer Werte führende Leerzeichen, Tabs und Zeilenvorschübe, erwartet dann eine Eingabe, die mit Enter abgeschlossen wird, und läßt diesen abschließenden Zeilenvorschub für die nächste Leseaktion »stehen«; *Readln* führt dieselbe Aktion durch und entfernt den abschließenden Zeilenvorschub nur dann aus dem Eingabepuffer, wenn er den Abschluß des letzten angeforderten Elements darstellt.

Bei der Eingabe von Strings werden die Unterschiede zwischen *Read* und *Readln* seit der Version 4.0 deutlich: In einem Programm, das zwei aufeinanderfolgende Strings mit *Read* anfordert, also

```
program StrRead;
var S1, S2: string[79];
begin
  Read(S1);
  Read(S2);
end.
```

liest der erste Aufruf von *Read* eine Eingabezeile bis zum abschließenden Zeilenvorschub und läßt ihn »stehen«. Der zweite Aufruf bekommt folglich als erstes diesen Zeilenvorschub zu sehen, ordnet der Variablen *S2* einen Nullstring zu und endet ohne eine Interaktion mit dem Benutzer. (Der einzige Unterschied

zwischen *Read* und *Readln* in der Version 3.0 besteht darin, daß *Read* den Zeilenvorschub nicht auf den Bildschirm ausgibt – dort würde das Programm also im gewünschten Sinne funktionieren).

Als Konsequenz dieses standardisierten Verhaltens sollten Programme ab der Versionen 4.0 deshalb grundsätzlich *Readln* anstelle von *Read* für Eingaben über die Tastatur verwenden.

Vordefinierte Bezeichner

In der Version 3.0 sind sämtliche vordefinierten Konstanten, Datentypen, Variablen, Prozeduren und Funktionen zu jedem Zeitpunkt verfügbar. Die Version 6.0 »verteilt« Bezeichner, die nicht dem Sprachstandard entsprechen, auf mehrere Standard-Units (*Dos*, *Crt*, *Printer*, *Overlay*, *Graph*, *Turbo3* und *Graph3*). Um die jeweiligen Bezeichner verwenden zu können, müssen Sie zuerst die entsprechende Unit über eine **uses**-Anweisung in Ihr Programm aufnehmen:

```
uses
  Crt, Dos, ....;
```

Tabelle A.2 gibt die Zusammenhänge zwischen Standard-Units und Bezeichnern wieder, die in der Version 3.0 vordefiniert sind:

Tabelle A.2 Units und vordefinierte Bezeichner

6.0-Unit	3.0-Bezeichner
Dos	*MsDos*, *Intr*, *Exec* (*Execute* in 3.0)
Crt	*KeyPressed*, *TextMode*, *Window*, *GotoXY*, *WhereX*, *WhereY*, *ClrScr*, *ClrEol*, *InsLine*, *DelLine*, *TextColor*, *TextBackground*, *LowVideo*, *NormVideo*, *Delay*, *Sound*, *NoSound* und alle Farb-Konstanten
Printer	*Lst*
Turbo3	*Kbd*, *CBreak*, *LongFileSize*, *LongFilePos*, *LongSeek*
Graph3	sämtliche Grafik- und Turtlegrafik-Routinen

Die Prozeduren *MsDos* und *Intr* der Version 3.0 arbeiten mit einem untypisierten Parameter, der innerhalb des Programms als Record definiert werden muß. In der Version 6.0 erwarten Sie dagegen eine Variable des (in *Dos* definierten) Typs *Registers*. Für *Intr* muß der erste Parameter in der Version 3.0 eine Konstante sein, die Version 6.0 erlaubt dagegen einen beliebigen Ausdruck, solange er einen Wert des Typs *Byte* ergibt.

Die Prozedur *Execute* der Version 3.0 wurde durch *Exec* (in der Unit *Dos*) ersetzt und erwartet keine Dateivariable, sondern den Namen eines Programms, dem zusätzlich Kommandozeilen-Parameter übergeben werden können.

Die Version 3.0 enthält zur Steuerung von Ein- und Ausgaben noch eine Reihe vordefinierter Bezeichner: *CrtExit*, *CrtInit*, *ConInPtr*, *ConOutPtr*, *ConStPtr*, *LstOutPtr*, *UsrInPtr*, *UsrOutPtr* und *ErrorPtr*. Die Version 6.0 ermöglicht die

Erstellung eigener Gerätetreiber (siehe Kapitel 14 »Interne Details« des Programmierhandbuches) und definiert diese Bezeichner nicht.

Das Gerät *Lst* ist über die Unit *Printer* implementiert, *Kbd* ist nach wie vor über *Turbo3* verfügbar. Die restlichen Geräte (*Aux*, *Con*, *Trm* und *Usr*) sind in der Version 6.0 nicht mehr als vordefinierte Bezeichner vorhanden – sie können aber ebenfalls über selbstgeschriebene Treiber definiert werden, wobei dem Programmierer erheblich mehr Möglichkeiten zur Verfügung stehen.

Der Aufruf der Funktion *TextMode* ohne Parameter setzt in der Version 3.0 wieder den Textmodus, der vor der letzten Umschaltung zur Grafik aktiv war. In der Version 6.0 muß dazu die (in der Unit *Crt* definierte) Variable *LastMode* verwendet werden, also

```
TextMode(LastMode);
```

Die Funktionen *MemAvail* und *MaxAvail* der Version 3.0 liefern einen Integer zurück, der die Größe des verfügbaren Speichers in Paragraphen (Einheiten zu je 16 Bytes) angibt, und sind nach wie vor über die Unit *Turbo3* verfügbar. Die gleichnamigen Funktionen der Version 6.0 haben den Ergebnistyp *LongInt* und arbeiten mit der Einheit Byte.

Die Routinen *FileSize*, *FilePos*, *FileSeek* (Ergebnistyp *Integer*) und ihre Gegenstücke *LongFileSize*, *LongFilePos* und *LongSeek* (Ergebnistyp *Real*) der Version 3.0 wurden ab der Version 4.0 vereinheitlicht: *FileSize*, *FilePos* und *Seek* liefern einen *LongInt* zurück, *Seek* erwartet einen Parameter dieses Typs. Programme, die mit *FileSize*, *FilePos* und *Seek* arbeiten, müssen entsprechend modifiziert werden, die »Long«-Versionen dieser Routinen sind nach wie vor über die Unit *Turbo3* verfügbar.

Die in der Version 3.0 von der Funktion *IOResult* zurückgelieferten Fehlercodes sind Turbo Pascal-spezifisch; ab der Version 4.0 werden dagegen die Standardcodes von MS-DOS erzeugt. (Die Unit *Turbo3* enthält eine eigene Version von *IOResult*, die die alten Fehlernummern verwendet). Die alte Version von *IOResult* liefert bei mehreren aufeinanderfolgenden Fehlern den Code des ersten Fehlers zurück, die neue Version dagegen den Code des tatsächlich zuletzt aufgetretenen Fehlers.

Die Version 3.0 definiert eine globale Variable namens *CBreak*, deren Wert festlegt, ob Ein- und Ausgaben über Ctrl Break abgebrochen werden können. Diese Variable hat in der Version 6.0 den Namen *CheckBreak*, der Bezeichner *CBreak* ist aber nach wie vor über die Unit *Turbo3* verfügbar.

Programmiertechniken

Addr-Funktion

In der Version 3.0 liefert die Funktion *Addr* die Adresse beliebiger Variablen zurück. Auch wenn diese Funktion in der Version 6.0 nach wie vor unterstützt wird, sollte an ihrer Stelle der Adreß-Operator @ verwendet werden:

```
Zeiger_Var := Addr(Variable);   { alte Version }
Zeiger_Var := @Variable;    { neue Version }
```

BCD-Arithmetik

In der Version 3.0 ist ein weiterer Compiler verfügbar, der mit binär codierten Dezimalzahlen arbeitet. Die entsprechenden Datentypen müssen in der Version 6.0 durch die IEEE-Gleitkommatypen ersetzt werden (die ihrerseits den Modus {*$N+*} und damit entweder einen numerischen Coprozessor oder den Emulator (Compiler-Befehl: {*$E+*}) voraussetzen).

Der Datentyp *Comp*, der ganzzahlige Werte mit jeweils 8 Byte speichert und speziell für Anwendungen vorgesehen ist, bei denen keine Rundungsfehler auftreten dürfen, dürfte für die Übernahme von BCD-Zahlen am besten geeignet sein. (Ein entsprechendes Beispiel dazu finden Sie auf den Originaldisketten).

Gleitkomma-Konstanten müssen in der Version 6.0 mit mindestens einer Ziffer vor dem Dezimalpunkt angegeben werden – eine Angabe wie *.1234* für den Wert *0.1234* ergibt eine entsprechende Fehlermeldung des Compilers.

Char-Typen und Strings

In der Version 3.0 ist die Verwendung eines Strings der Länge 1 als **case**-Selektor zulässig, in der Version 6.0 dagegen nicht: hier müssen Sie anstelle von *String_Var* das Zeichen selbst angeben (*String_Var[1]*).

Analoges gilt für die Kompatibilität von Strings der Länge 1 und Variablen des Typs *Char*. In der Version 3.0 ist sowohl eine direkte Zuweisung als auch die Anwendung der Funktion *Copy* legal:

```
var
  Ch : Char;
  S : string[10];
begin
  S := 'a';
  Ch := S;   { nur in Version 3.0 möglich }

  S := 'abc';
  Ch := Copy(S,2,1);  { dito }
  ...
```

Keine dieser beiden Operationen übersteht die Typ-Prüfung des Compilers in der Version 6.0. Nach wie vor möglich (weil mit dem ANSI-Standard von Pascal konform) sind Zuweisungen in umgekehrter Richtung (*S:=Ch*), sowie die Indizierung einzelner Zeichen eines Strings als *Char* (*Ch := S[2]*).

Close-Prozedur

In der Version 3.0 erzeugt der Aufruf der Prozedur *Close* für eine bereits geschlossene Datei keinen Fehler – es passiert einfach gar nichts. Die *Close*-Prozedur der Version 6.0 erzeugt in derartigen Fällen einen Laufzeitfehler, der über {*$I-*} abgefangen und mit *IOResult* ermittelt werden kann.

CSeg und DSeg

DSeg (Funktion):Unterschiede zur Version 3.0;In der Version 3.0 können *CSeg* und *DSeg* in Deklarationen mit **absolute** verwendet werden, also

```
var
  Parameter: string[127] absolute CSeg:$80;  { nur Version 3.0 }
```

Da ab der Version 4.0 die Segment-Zugehörigkeit durch den Linker festgelegt wird, sind derartige Deklarationen nicht erlaubt.

Recordgrößen untypisierter Dateien

Unvollständige Records untypisierter Dateien werden in der Version 6.0 ignoriert: *BlockRead* liest nur vollständige Records, *FileSize* liefert die Anzahl der vollständigen Records einer Datei zurück.

Eingaben von Integern

In der Version 3.0 läßt *Readln* den Wert einer angegebenen Integervariablen unverändert, wenn überhaupt keine Zeichen eingegeben werden. Eine Befehlsfolge wie

```
var
  I : Integer;
begin
  I := 30;
  Write('Geben Sie einen Wert ein: '); Readln(I);
  ...
```

läßt also *I* unverändert auf dem Wert 30, wenn der Benutzer anstelle einer Eingabe nur auf Enter drückt. In der Version 6.0 wiederholt *Readln* die Abfrage der Tastatur dagegen solange, bis auch tatsächlich etwas eingegeben wird.

Schleifenvariablen für for

In der Version 3.0 spielt es keine Rolle, wo und wie die Laufvariable einer **for**-Schleife deklariert ist. In der Version 6.0 muß eine derartige Variable entweder global zum Programm oder lokal zur jeweiligen Routine sein. Das folgende Programmfragment wird deshalb vom Compiler als fehlerhaft ausgewiesen:

```
procedure Aussen;
var
  I : Integer;
  procedure Innen;

  begin
   for I := 1 to 10 do   { I ist weder global noch lokal zu "Innen" }
     Writeln(I);
  end;  { der Prozedur Innen }
begin  { der Prozedur Aussen }
  ...
end;  { der Prozedur Aussen }
```

Identische und kompatible Typen

In der Version 3.0 sind identisch deklarierte Typen auch dann zuweisungskompatibel, wenn sie voneinander getrennt deklariert sind. Das folgende Programmfragment ist also legal:

```
var
  A:  ^Integer;
  B:  ^Integer;

begin
  ...
  A := B;   { nur in Version 3.0 erlaubt }
  ...
```

Die Version 6.0 nimmt eine strengere Typenprüfung vor und reagiert auf derartige Formulierungen mit einer Fehlermeldung. Damit zwei Variablen zuweisungskompatibel sind, müssen Sie entweder zusammen deklariert oder über einen gemeinsamen Typ vereinbart sein:

```
type
  IntPtr = ^Integer;
var
  A, B:  ^Integer;   { erste Möglichkeit }
  C  : IntPtr;
  D  : IntPtr;       { zweite Möglichkeit }

begin
  ...
  A := B;  C:= D;   { beide Zuordnungen sind legal }
  A := C;    { wird vom Compiler nicht akzeptiert! }
  ...
```

Integerwerte

In der Version 3.0 ist die direkte Zuweisung des Wertes -32768 an eine Variable des Typs *Integer* nur über die Angabe der Konstanten *$8000* möglich. Die Version 6.0 erlaubt die Angabe von -32768 und engt gleichzeitig mögliche Zuweisungen an *Integer*-Variablen auf den korrekten Wertebereich (-32768 ..32767) ein: Für Werte größer/gleich $8000 müssen Sie den Typ *Word* oder *LongInt* verwenden.

Typisierte Konstanten

Der Compiler der Version 3.0 setzt typisierte Konstanten in das Code-Segment des Programms, in der Version 6.0 werden sie dagegen im Daten-Segment gespeichert.

Labels

In der Version 3.0 werden nicht verwendete Labels einfach ignoriert. Die Version 6.0 liefert eine Fehlermeldung, wenn ein Bezeichner als **Label** definiert ist, aber nirgendwo innerhalb des Programms deklariert wird.

Direkte Kontrolle der Tastatur

In der Version 3.0 wird die Geräteeinheit *Kbd* in der Anweisung *Read*(*Kbd, Ch*); für das direkte Lesen der Tastatur ohne automatische Ausgabe auf den Bildschirm verwendet. *Kbd* ist ab der Version 4.0 nur noch in der Unit *Turbo3* definiert. Aufrufe der Form *Read*(*Kbd,Ch*) werden durch die neue Funktion

ReadKey ersetzt, mit der die Prüfung auf Spezialtasten wesentlich vereinfacht wird:

```
...
Ch := ReadKey;   { liest ein Zeichen }
if Ch := #0 then
       begin            { Spezialtaste: }
          Ch := ReadKey; { Scancode lesen }
          .....  { Bearbeitung }
        end
     else ....   { normale Taste }
```

Puffergröße von Textdateien

In der Version 3.0 kann bei der Deklaration einer Datei-Variablen als *Text* optional eine Puffergröße angegeben werden:

```
var
  f: Text[4096];  { 4096 Bytes Puffer -- nur Version 3.0 }
```

Die Version 6.0 erwartet eine explizite Deklaration des Puffers als Variable und eine Zuordnung zu der Datei-Variablen über die Prozedur *SetTextBuf*:

```
var
  F: Text;
  Puffer: array[1..4096] of Char;
begin
  Assign(F, 'Datei_x.TXT');
  SetTextBuf(F, Puffer);
  Reset(F);
  ....
```

Erweiterungen und Verbesserungen

Umwandlung von Variablentypen

In der Version 3.0 muß die Funktion *Move* zum Kopieren von Daten verwendet werden, wenn die beiden beteiligten Datenstrukturen nicht zuweisungskompatibel sind.

Die einzige Ausnahme von dieser Regel stellen einfache ordinale Typen (*Char, Byte, Boolean, Integer* und Aufzählungstypen) dar, bei denen eine explizite Umwandlung möglich ist:

```
type
  Puffer = array[0..5] of Byte;
  Karten = Kreuz, Pik, Herz, Karo;
var
  X_Real: Real; X_Array : ^Puffer;
  X: Integer; X_Karten: Karten;
begin
```

```
New(X_Array);
...
Move(X_Array^, X, SizeOf(X);   { geht nur über Move }

X := Integer(Karten);
X := Byte('A');        { direkt möglich }
...
```

Ab der Version 4.0 wurde die Typ-Umwandlung auf sämtliche Datentypen erweitert – die einzige Bedingung ist, daß Quelle und Ziel einer Umwandlung dieselbe Größe haben müssen.

```
type
  Puffer = array[0..5] of Byte;
var
  X_Real: Real; X_Array : ^Puffer;

begin
  New(X_Array);
   ...
  X := Real(X_Array^);
  ...
```

Integer-Typen

In der Version 3.0 stehen für Berechnungen mit ganzzahligen Werten die Typen *Byte* (0..255, 1 Byte) und *Integer* (-32768..32767, 2 Bytes) zur Verfügung. Die Version 6.0 bietet die zusätzlichen Typen *ShortInt* (-128..127, 1 Byte), *Word* (0..65535, 2 Bytes) und LongInt (-2147483648..2147483647, 4 Bytes).

Real-Typen

In der Version 3.0 steht für Berechnungen mit Gleitkommazahlen ausschließlich der Typ *Real* zur Verfügung. Im Modus {*$N+*} bietet die Version 6.0 die zusätzlichen Typen *Single* (4 Bytes), *Double* (8 Bytes), *Extended* (10 Bytes) und *Comp* (8 Bytes, für Operationen mit ganzzahligen Werten).

String-Variablen

In der Version 3.0 muß jede Deklaration des Typs *String* eine explizite Längenangabe enthalten. Um Strings als Parameter zu verwenden, ist darüber hinaus die Definition eigener Typen erforderlich:

```
type
  BigStr = string[255];
var
  Name : string[20];
  S    : BigStr;
...
procedure Was_auch_immer(T : BigStr);
  ...
```

Seit der Version 4.0 können Variablen als *String* ohne Längenangabe dekla-

riert werden, die einer Deklaration als *String*[*255*] entsprechen. Mit *String* ist hier auch die Deklaration formaler Parameter möglich:

```
var
  S : string;
...
procedure X_beliebig(T : String);
```

Auswertung boolscher Ausdrücke

In der Version 3.0 werden zusammengesetzte boolsche Ausdrücke immer vollständig bewertet, auch wenn das Ergebnis nach der Auswertung des ersten Teils bereits feststeht. Aus diesem Grund erzeugt eine Prüfung wie

```
if (B = 0) or (A/B =X) then
```

einen Laufzeitfehler, wenn *B* den Wert 0 hat – die Division wird durchgeführt, obwohl die erste Bedingung (*B = 0*) bereits den Wert *True* ergeben hat und sich am Gesamtergebnis des Ausdrucks (*True*) nichts mehr ändern kann. Die Version 6.0 arbeitet mit einem *Kurzschlußverfahren*, das die Auswertung derartiger Ausdrücke abbricht, sobald das Ergebnis feststeht. (Über den Compiler-Befehl {*$B+*} läßt sich optional festlegen, daß auch hier mit vollständiger Bewertung gearbeitet wird).

Laufzeitfehler

In der Version 3.0 können Laufzeitfehler nur über Zuweisungen an die globale Variable *ErrorPtr* abgefangen werden. Die Version 6.0 verwendet anstelle von *ErrorPtr* eine Variable namens *ExitProc*, über die sich nicht nur eine Fehlerbehandlungsroutine, sondern auch der »normale« Abschluß eines Programms festlegen läßt (siehe Kapitel 14 »Interne Details« des Programmierhandbuches).

Interrupt-Behandlung

In der Version 3.0 können Behandlungsroutinen für Interrupts nur in Maschinensprache (und via **inline**) codiert werden. In der Version 6.0 lassen sich derartige Routinen über eine Deklaration mit **interrupt** mit normalen Pascal-Befehlen und **inline**-Anweisungen in gemischter Form schreiben.

Direkter Speicherzugriff

In der Version 3.0 stehen die vordefinierten Bezeichner *Mem* und *MemW* für das direkte Lesen und Setzen von Speicherzellen zur Verfügung, die in der Version 6.0 um das Pseudo-Array *MemL* mit dem Komponententyp *LongInt* erweitert sind. Die Funktion *MemW* hat in der Version 6.0 den Ergebnistyp *Word*.

Maschinencode

In der Version 3.0 steht nur die Anweisung **inline** zum Einfügen von Maschinencode innerhalb von Funktionen, Prozeduren und des Hauptprogramms zur Verfügung. Die Version 6.0 erweitert die Anwendungsmöglichkeiten von **inline** auf komplette Prozeduren und Funktionen, die der Compiler (als eine Art Makros) direkt in den laufenden Code einfügt.

In der Version 3.0 müssen extern eingebundene Maschinenprogramme als .BIN-Dateien codiert sein und werden innerhalb des Programms als relativer Wert (»Offset«) zum Dateianfang adressiert. Die Version 6.0 arbeitet mit .OBJ-Dateien, deren Bezeichner einfach als **external** deklariert sind; die Berechnung der Adressen erfolgt automatisch durch den Linker.

Prozedur- und Funktionsaufrufe

In der Version 3.0 wird ausschließlich mit Aufrufen des Typs *near* gearbeitet, wodurch der Umfang des Programms auf ein Segment (64 KByte Code) begrenzt ist. Die Version 6.0 codiert Aufrufe und Rücksprünge je nach Bedarf als *near* oder *far*, über den Compiler-Befehl {$F+} lassen sich *far*-Aufrufe erzwingen.

Fehlersuche

In der Version 3.0 sind keine Hilfestellungen für die Fehlerbereinigung enthalten. Die Version 6.0 beinhaltet nicht nur einen integrierten Debugger zur Fehlersuche auf der Quelltext-Ebene, sondern kann auch zusätzlich Informationen für *Turbo Debugger* erzeugen, mit dem sich die Programmausführung auf der Ebene der Maschinensprache verfolgen und manipulieren läßt (siehe Kapitel 6 »Fehlersuche und der integrierte Debugger«).

Die Units Turbo3 und Graph3

Diese beiden »Kompatibilitäts-Units« sind in Kapitel 12 »Standard-Units« des Programmierhandbuches so ausführlich besprochen, daß wir an dieser Stelle nur noch einmal eine kurze Zusammenfassung geben:

Turbo3 und *Graph3* sind zwar Teil der Unit-Bibliothek von Turbo Pascal, sind jedoch nicht in der Datei TURBO.TPL enthalten – die Aufnahme der beiden Kompatibilitäts-Units in TURBO.TPL ist mit dem Zusatzprogramm TPUMOVER möglich, das in Kapitel 10 »Die Zusatzprogramme« beschrieben ist. Um TURBO3 und GRAPH3 in ein Programm aufzunehmen, müssen beide Dateien im richtigen Verzeichnis abgelegt oder in TURBO.TPL enthalten sein. Für die eigentliche Einbindung ist lediglich eine entsprechende **uses**-Anweisung erforderlich, die allerdings auch die Unit *Crt* enthalten muß:

```
uses Crt, Turbo3;          { für Turbo3 }
uses Crt, Graph3;          { für Graph3 }
uses Crt, Turbo3, Graph3;  { für beide Units }
```

Hinweis: Turbo3 und *Graph3* sind ausschließlich zur Übernahme existierender Programme vorgesehen und sollten auch nur zu diesem Zweck verwendet werden.

Turbo3

Diese Unit deklariert verschiedene Bezeichner der Laufzeitbibliothek, die ab der Version 4.0 nicht mehr vorhanden sind, und einige Routinen, die verändert bzw. ersetzt wurden:

Kbd: Die Geräteeinheit zum direkten Lesen der Tastatur; in der Version 6.0 durch die Funktion *ReadKey* (in der Unit *Crt*) ersetzt.

CBreak: Eine undokumentierte boolsche Variable, deren Wert bestimmt, ob ein Programm mit Ctrl Break unterbrochen werden kann; in der Version 6.0 durch die (in Kapitel 12 »Standard-Units« des Programmierhandbuches dokumentierte) Variable *CheckBreak* ersetzt;

MemAvail: Liefert die Gesamtmenge des freien Speicherplatzes auf dem Heap in Paragraphen (Einheiten zu 16 Bytes) als *Integer* zurück; ab der Version 4.0 durch eine Routine gleichen Namens ersetzt, die aber den Ergebnistyp *LongInt* hat und den freien Platz in Bytes zurückliefert.

MaxAvail: Liefert den Umfang des größten freien Bereiches auf dem Heap in Paragraphen (Einheiten zu 16 Bytes) als *Integer* zurück; ab der Version 4.0 durch eine Routine gleichen Namens ersetzt, die aber den Ergebnistyp *LongInt* hat und den freien Platz in Bytes zurückliefert.

LongFileSize: Liefert die Größe (Komponentenzahl) einer Datei als *Real* zurück; in der Version 6.0 durch die Funktion *FileSize* mit dem Ergebnistyp *LongInt* ersetzt.

LongFilePos: Liefert die momentane Position innerhalb einer Datei (Recordnummer) als *Real* zurück; in der Version 6.0 durch die Funktion *FilePos* mit dem Ergebnistyp *LongInt* ersetzt.

LongSeek: Setzt die momentane Position (Recordnummer) innerhalb einer Datei und erwartet ein Argument vom Typ *Real*; in der Version 6.0 durch die Prozedur *Seek* ersetzt, die mit Argumenten des Typs *LongInt* arbeitet.

IOResult: Liefert den Fehlerstatus der letzten Ein-/Ausgabeoperation zurück, wobei Turbo Pascal-spezifische (und mit der CP/M-Version 3.0 identische) Fehlercodes benutzt werden; in der Version 6.0 durch eine Routine gleichen Namens ersetzt, die die Fehlercodes von MSDOS verwendet. Nach der Aufnahme von *Turbo3* liefern Aufrufe von *IOResult* wieder die »alten« Fehlercodes.

NormVideo, LowVideo und HighVideo: Drei Standard-Prozeduren zum Setzen von (Text-)Bildschirmfarben, die in der Version 6.0 auf andere Weise arbeiten (siehe Kapitel 1 »Verzeichnis der Routinen« des Referenzhandbuches). Nach der Aufnahme von *Turbo3* setzen diese drei Prozeduren wieder dieselben Farben wie in der Version 3.0:

	Mono, S/W	Farbmonitor
LowVideo	hellgrau	hellgrau
NormVideo	weiß	gelb
HighVideo	weiß	g elb

Graph3

Diese Unit enthält alle Grafik-Routinen der Version 3.0 inklusive der Turtlegrafik. Eine Auflistung der entsprechenden Konstanten, Variablen, Prozeduren und Funktionen finden Sie in Kapitel 12 »Standard-Units« des Programmierhandbuches; ansonsten verweisen wir auf das Kapitel 19 des Handbuches von Turbo Pascal 3.0.

Die Version 6.0 enthält die Unit *Graph* – ein komplettes Grafikpaket, das dazu (fast) unabhängig von der verwendeten Hardware-Ausrüstung arbeitet. Die Routinen und Bezeichner von *Graph* sind ebenfalls in Kapitel 12 »Standard-Units« des Programmierhandbuches kurz beschrieben, Kapitel 1 »Verzeichnis der Routinen« des Referenzhandbuches erläutert jede dieser Routinen im Detail.

Die Konvertierung im Detail

Auch nach der Aufnahme der »Kompatibilitäts-Units« *Turbo3* und *Graph3* werden Sie speziell bei größeren Programmen um einige manuelle Änderungen nicht herumkommen. Die folgenden Abschnitte beschreiben das Wie und Wo. Wir haben dabei eine Einteilung in drei »Klassen« vorgenommen:

EMPFOHLEN: Die Veränderung nützt die neuen Möglichkeiten der Version 6.0 aus – das Programm wird schneller, kürzer, übersichtlicher... Ob Sie diese Veränderungen vornehmen oder nicht, hängt von Ihrem persönlichen Geschmack ab.

NOTWENDIG: Veränderungen dieser Art sollten wirklich ausgeführt werden. Es ist möglich, daß Sie (unter Umständen mit Tricks) darum herumkommen – allerdings auf eigene Gefahr.

UNUMGÄNGLICH: Bei diesen Punkten gibt es keine Diskussion und keine Ausweichmöglichkeiten – entweder Sie führen sie aus oder Ihr Programm läßt sich nicht compilieren bzw. arbeitet nicht korrekt.

Vordefinierte Bezeichner

Die Versionen ab 4.0 unterstützen nicht alle vordefinierten Bezeichner (Konstanten, Typen, Variablen, Prozeduren und Funktionen) der Version 3.0. In einigen Fällen sind diese Bezeichner ersetzt worden, andere finden sich in Standard-Units wieder, und ein paar haben wir überhaupt nicht mehr zur Verfügung gestellt.

- Verwenden Sie die neuen Routinen der Standard-Units wie *ReadKey* anstelle von *Read*(*Kbd, Ch*). (EMPFOHLEN)
- Benutzen Sie die Deklarationen aus den entsprechenden Standard-Units für

Bezeichner, die in der Version 3.0 grundsätzlich verfügbar waren. Die Routinen *Intr* und *MsDos* gehören beispielsweise in der Version 6.0 nicht mehr zum Sprachstandard, sondern sind in einer separaten Unit (*Dos*) deklariert. Entsprechendes gilt für das Gerät *Lst*, dessen Deklaration sich in der Unit *Printer* findet. (UNUMGÄNGLICH)

Datentypen

Die Version 6.0 führt eine ganze Reihe neuer Datentypen zusammen mit Routinen ein, die diese Typen benutzen. In vielen Fällen werden »Programmiertricks« aus alten Zeiten damit unnötig gemacht.

■ Verwenden Sie eine explizite Typ-Umwandlung anstelle von *Move* für Operationen mit ansonsten inkompatiblen Variablentypen. Die folgenden beiden »Zuweisungen« haben in der Version 6.0 dasselbe Ergebnis:

```
Move(PufferZeiger^,RealVar, SizeOf(RealVar));
RealVar := Real(PufferZeiger^);
```

Die zweite Zuweisung ist nicht nur übersichtlicher, sondern benötigt auch weniger Code und wird schneller ausgeführt. Explizite Typ-Umwandlungen können überall dort eingesetzt werden, wo die Ziel-Variable dieselbe Größe wie die Quell-Variable hat. (EMPFOHLEN)

■ Verwenden Sie die neuen Datentypen seit der Version 4.0 an Stellen, wo ansonsten mit Programmiertricks gearbeitet werden muß. Die Typen *LongInt* und *Word* ersetzen *Integer* bei großen bzw. vorzeichenlosen Werten, der generische Typ *Pointer* ersetzt ^*Byte* und ^*Char*, der Typ *string* ersetzt Deklarationen mit vorgegebener Stringlänge und entspricht *string*[*255*]. (EMPFOHLEN)

■ Beachten Sie, daß die Version 6.0 hexadezimale Konstanten als *Word* behandelt (und nicht als *Integer*): die Angabe $FFFF steht also für 65535 und nicht für -1. Variablen, denen hexadezimale Konstanten zugewiesen werden, sollten folglich als *Word* deklariert sein. (NOTWENDIG)

■ Entsprechendes gilt für die Zuweisung des Wertes -32768 an Variablen des Typs *Integer*, die in der Version 3.0 nur über die Angabe von $8000 möglich war. Da die Version 6.0 die Konstante $8000 als vorzeichenlos (und damit als +32768) behandelt, meldet der Compiler derartige Zuweisungen als Fehler. Ersetzen Sie die Angabe $8000 durch -32768 oder geben Sie den entsprechenden Variablen den Typ *Word*. (NOTWENDIG)

■ Verwenden Sie die (in der Version 6.0 optimierten) Bibliotheksroutinen *Length* und *Copy* anstelle direkter Zugriffe auf die interne Struktur von Strings wie *Ord*(*StringVar*[*0*]) oder als **absolute** deklarierte *Byte*-Variablen, deren Adresse mit *StringVar*[*0*] zusammenfällt. (NOTWENDIG)

■ Beachten Sie, daß die Version 6.0 eine wesentlich strengere Typenprüfung für *Char*-Variablen, Strings und *Char*-Arrays durchführt. Zuweisungen der Form

```
CharVar := StringVar;
```

werden auch dann als fehlerhaft zurückgewiesen, wenn *StringVar* als *string[1]* deklariert ist. Zuweisungen von *Char*-Arrays an Strings sind nach wie vor erlaubt:

```
StringVar := Char_Array;   { ist legal }
Char_Array := StringVar;   { FALSCH! }
```

Zuweisungen von Strings an *Char*-Arrays werden dagegen vom Compiler als Typ-Verletzung zurückgewiesen. (UNUMGÄNGLICH)

■ Die strengere Typenprüfung der Version 6.0 erstreckt sich auch auf identische und kompatible Typen (vgl. Kapitel 4 »Variablen« des Programmierhandbuches und »Programmiertechniken« im ersten Teil dieses Anhangs). Zuweisungskompatibilität ist nur dann gegeben, wenn beide beteiligten Variablen entweder gemeinsam oder über einen zuvor definierten Typ deklariert werden. Bei einer Deklaration wie

```
var
  A: ^Integer;
  B: ^Integer;
```

wird keiner der Grundtypen von Turbo Pascal verwendet, folglich ergeben Zuweisungen wie *A:=B* eine Fehlermeldung des Compilers. Die Definition eines gemeinsamen Typs, also

```
type
  IntPtr := ^Integer;
var
  A: IntPtr;
  B: IntPtr;
```

oder die gemeinsame Deklaration beider Variablen wie in

```
var
  A, B: ^Integer;
```

erzeugt *identische* Typen und stellt damit auch die Zuweisungskompatibilität her. Der erste Lösungsweg ist dabei der bessere, weil er die Deklaration weiterer zuweisungskompatibler Variablen zu einem späteren Zeitpunkt erlaubt. (UNUMGÄNGLICH)

■ Die BCD-Datentypen werden von der Version 6.0 nicht unterstützt, die Routine *Form* ist nicht definiert. Verwenden Sie den Datentyp *LongInt*; falls Ihr Computer mit einem Coprozessor ausgerüstet ist, sollten Sie den Modus {*$N+*} aktivieren und den (IEEE-)Typ *Comp* benutzen, der ganzzahlige Werte mit 8 Bytes speichert. Die Dateien BCDREAL.PAS und BCD.PAS auf den Originaldisketten demonstrieren die Möglichkeiten zur Konvertierung von BCD-Programmen. (UNUMGÄNGLICH)

Sprachstruktur

Die Version 6.0 führt einige Beschränkungen und Erweiterungen ein, um mit der ANSI-Definition von Pascal konform zu gehen. Die wichtigsten Veränderungen gegenüber der Version 3.0 sind:

- Kurzschlußverfahren bei der Auswertung zusammengesetzter boolscher Ausdrücke: In der Version 6.0 wird die Bewertung eines Ausdrucks abgebrochen, sobald das Ergebnis des Gesamtausdrucks feststeht.

  Ein Ausdruck wie

  ```
  if expr1 and expr2 then ...
  ```

  hat unabhängig von *expr2* das Ergebnis *False*, wenn *expr1* den Wert *False* ergibt. Bei Verwendung des Kurzschlußverfahrens wird *expr2* in diesem Fall überhaupt nicht mehr bewertet.

  Damit werden einige Probleme wie in

  ```
  if(B <> 0) and(A div B > 5) then ...
  ```

  umgangen, bei denen sich ansonsten ein Fehler durch die Prüfung selbst ergäbe. Wenn *expr2* im zuvor gezeigten Beispiel allerdings einen Funktionsaufruf enthält, dann wird diese Funktion auch nur dann aufgerufen, wenn das Gesamtergebnis des Ausdrucks nach der Bewertung von *expr1* noch nicht feststeht.

  Als Standardvorgabe arbeitet Turbo Pascal mit dem Kurzschlußverfahren. Über den Compiler-Schalter {*$B+*} zu Beginn eines Quelltextes läßt sich festlegen, daß auch zusammengesetzte Ausdrücke vollständig bewertet werden. (EMPFOHLEN)

- Wie durch den ANSI-Standard festgelegt, erlaubt die Version 6.0 von Turbo Pascal nur globale oder lokale Deklarationen von **for**-Laufvariablen. Für eine Anweisung wie

  ```
  for Index := 0 to 10 do ...
  ```

  muß die Variable *Index* entweder global zum gesamten Programm oder lokal zu der Routine deklariert sein, in der auch die **for**-Anweisung steht. (UNUMGÄNGLICH)

Ein- und Ausgabe

Die Version 6.0 enthält einige starke Veränderungen bei Ein-/Ausgabeoperationen gegenüber der Version 3.0, die hauptsächlich auf eine Angleichung an den ANSI-Standard zurückzuführen sind.

- Die Anweisung *Read*(*IntVar*) wiederholt die Eingabeanforderung solange, bis tatsächlich eine Eingabe erfolgt. In der Version 3.0 hält *Read* die Eingabe auch dann für beendet, wenn der Benutzer nur auf Enter drückt, und läßt in diesem Fall den vorherigen Wert von *IntVar* unverändert. Entsprechende Prüfungen

innerhalb des Programms können also in der Version 6.0 entfallen. (EMPFOHLEN)

- Achten Sie bei der Speicherung von Realwerten in Dateien auf die Unterschiede zwischen dem Typ *Real* (6 Bytes, kompatibel mit der Version 3.0) und den im Modus {*$N+*} unterstützten IEEE-Datentypen *Single*, *Double*, *Extended* und *Comp*. Für Berechnungen und Zwischenergebnisse ist der Typ *Extended* in den meisten Fällen die beste Wahl – für die Speicherung in Dateien dagegen nur, wenn auch alle anderen Systeme, die diese Dateien eventuell benutzen, mit einem Coprozessor ausgerüstet sind. (NOTWENDIG)

- Die Deklaration einer Textpuffergröße zusammen mit einer Datei-Variablen des Typs *Text* wird nicht mehr unterstützt und muß durch einen entsprechenden Aufruf der Standard-Prozedur *SetTextBuf* ersetzt werden. (NOTWENDIG)

- Die Begrenzung der Anzahl einzugebender Zeichen über eine Zuweisung an die Variable *BufLen* wird nicht mehr unterstützt (diese Variable ist in der Version 6.0 nicht deklariert). Zur Begrenzung von Eingaben müssen Sie eine eigene Routine schreiben. (UNUMGÄNGLICH)

- Die Prozedur *Close* erzeugt in der Version 6.0 einen Laufzeitfehler, wenn die angegebene Datei nicht offen ist oder nicht existiert – ein *Close* »für alle Fälle« am Ende eines Programms muß also entsprechend modifiziert bzw. mit {*$I-*} und {*$I+*} eingeklammert werden. (NOTWENDIG)

Organisation des Programms und Speicherbelegung

Eine Erläuterung des Unit-Konzepts finden Sie in Kapitel 3 »Units, Overlays und andere Geheimnisse« die technischen Hintergründe sind in Kapitel 14 »Interne Details« des Programmierhandbuches besprochen – wir beschränken uns deshalb auf die Feststellung, daß über Units Programme mit mehr als 64 KByte Code-Umfang möglich sind. Dementsprechend groß sind die Unterschiede in der Organisation eines Programms und seiner Speicherbelegung. Bei der Übernahme großer Programme sollten Sie die folgenden Punkte berücksichtigen:

- Konvertieren Sie existierende Funktions- und Prozedurbibliotheken von Include-Dateien in Units. (Dieser Schritt ist nur dann unbedingt notwendig, wenn Sie innerhalb Ihrer Hauptprogramme sonst mit Overlays arbeiten müßten). Die Konvertierung in Units verkürzt nicht nur die Compilierungszeiten drastisch, sondern hat auch noch den Vorteil, daß ganze Bibliotheken ohne den Quelltext weitergegeben werden können. (EMPFOHLEN)

- In der Version 6.0 kann die Größe des Stack sowie die Minimal- und Maximalgröße des Heap über den Compiler-Befehl {*$M*} festgelegt werden, der die folgende Syntax hat:

```
{$M Stackgröße, Heap_Minimal, Heap_Maximal}
```

Alle drei Werte werden in Byte angegeben, die Standardvorgaben des Com-

pilers sind {*$M 16384,0,655360*}. Alternativ zu einem {*$M*}-Befehl lassen sich diese Größen auch über das Menü *O/C/Memory sizes* bzw. den Kommandozeilen-Parameter `/M`festlegen. (EMPFOHLEN)

■ Overlays werden in der Version 6.0 nicht mehr auf der Basis einzelner Routinen, sondern *en bloc* als Units deklariert – Sie müssen also sämtliche als **overlay** deklarierten Routinen in einer oder mehreren Units zusammenfassen. (UNUMGÄNGLICH)

■ Beachten Sie, daß die Funktionen *MemAvail* und *MaxAvail* in der Version 6.0 den Ergebnistyp *LongInt* haben und die Größe des verfügbaren Speicherplatzes in Bytes zurückliefern. Gegebenenfalls können Sie hier allerdings die Unit *Turbo3* verwenden, die die alten Versionen dieser beiden Funktionen implementiert. (NOTWENDIG)

Compiler-Befehle und Fehlerprüfung

Sowohl Compiler-Befehle als auch die Fehlercodes sind in der Version 6.0 stark verändert worden. Eine kurze Zusammenfassung:

■ Die Standardvorgabe der Version 6.0 ist das Kurzschlußverfahren bei der Bewertung zusammengesetzter logischer Ausdrücke ({*$B+*}). Die Version 3.0 bewertet dagegen alle Ausdrücke vollständig, was dem Compiler-Befehl {*$B-*} zu Beginn des Programms entspricht. (EMPFOHLEN)

■ Die Überprüfung von Bereichen ist in der Version 6.0 standardmäßig abgeschaltet ({*$R-*}). Die Version 3.0 aktiviert diesen Schalter standardmäßig, was dem Compiler-Befehl {*$R+*} zu Beginn des Programms entspricht. (NOTWENDIG)

■ Prüfen Sie *sämtliche* Reaktionen des Programms auf Funktionsergebnisse von *IOResult* – speziell dann, wenn nicht nur mit dem Wert 0 verglichen wird. Definieren Sie alle Fehlercodes als globale Konstanten. (NOTWENDIG)

■ Prüfen Sie *sämtliche* Compiler-Befehle, speziell {*$B*}, {*$D*} und {*$F*}. Diese drei Befehle akzeptiert der Compiler auch in der Version 6.0 – allerdings haben sie hier eine andere Wirkung. (UNUMGÄNGLICH)

■ Die vordefinierte Variable *ErrorPtr* wird von der Version 6.0 nicht mehr unterstützt und muß durch *ExitProc* funktional ersetzt werden (siehe Kapitel 14 »Interne Details« des Programmierhandbuches). (UNUMGÄNGLICH)

■ Der Compiler-Befehl {*$I Dateiname*} darf in der Version 6.0 nicht innerhalb eines Blocks (d.h. zwischen einem **begin** und dem dazugehörigen **end**) gegeben werden; außerdem muß zwischen *$I* und dem Dateinamen mindestens ein Leerzeichen stehen. Modifizieren Sie Ihr Programm entsprechend. (UNUMGÄNGLICH)

Maschinensprache-Routinen

Die Anweisung **inline** ist im Gegensatz zur Version 3.0 erweitert worden: Neben einfachen Byte-Listen können nun auch Prozeduren und Funktionen als **inline** deklariert werden und stellen dann eine Art Makro dar (im Gegensatz zu einer Routine, die mit CALL aufgerufen und mit RET beendet wird). Der Abschnitt »Maschinensprache« in Kapitel 15 »Der integrierte Assembler und Einbinden von Assembler-Code« des *Programmierhandbuches* enthält eine ausführliche Besprechung der neuen Möglichkeiten.

- Kurze Folgen von Maschinenbefehlen, die mehrfach innerhalb eines Programms verwendet werden, sollten durch als **inline** deklarierte Routinen ersetzt werden. Der Compiler fügt derartige Routinen beim »Aufruf« ebenfalls direkt in den Code ein. (EMPFOHLEN)

- Umfangreichere Routinen sollten mit einem externen Assembler oder dem integrierten Assembler erstellt und in ersterem Fall als .OBJ-Dateien über den Compiler-Befehl {*$L*} in das Programm aufgenommen werden. (EMPFOHLEN)

- **inline** erlaubt in der Version 6.0 weder die Verwendung des Symbols * noch Bezüge auf Prozedur- und Funktionsnamen, weil die entsprechenden Adressen erst durch den Linker ermittelt werden. Um dennoch Bezüge auf Pascal-Routinen herzustellen, müssen Sie eine lokale Zeigervariable deklarieren, dieser Variablen die Routine (d.h. den Prozedur- oder Funktionsnamen) zuordnen und innerhalb der **inline**-Befehle auf diese Variable zugreifen. (UNUMGÄNGLICH)

- Externe Maschinenprogramme müssen neu als .OBJ-Dateien assembliert werden, das Dateiformat .BIN wird von der Version 6.0 nicht unterstützt. Für *Daten*, die im .BIN-Format gespeichert sind, kann allerdings das Programm BINOBJ zur Konvertierung verwendet werden. (UNUMGÄNGLICH)

- Typisierte Konstanten werden von der Version 6.0 nicht mehr im Code-Segment, sondern im Daten-Segment (*DSeg*) gespeichert. Assemblerroutinen, die auf typisierte Konstanten zugreifen, müssen entsprechend geändert werden. (UNUMGÄNGLICH)

- Die Übergabe von Parametern und das Zurückliefern von Funktionsergebnissen arbeiten mit stark veränderten Regeln. Wir verweisen auch hier auf Kapitel 15 »Der integrierte Assembler und Einbinden von Assembler-Code« des Programmierhandbuches. (UNUMGÄNGLICH)

- Verschiedentlich haben sich Programmierer die Tatsache zunutze gemacht, daß die Version 3.0 bei der Übergabe von Ausdrücken des Typs *Char* und *Byte* das höherwertige Byte automatisch auf den Wert 0 setzt. In der Version 6.0 ist das höherwertige Byte bei allen Übergaben von Ausdrücken mit 8 Bit undefiniert. Maschinenprogramme, die eine derartige Initialisierung voraussetzen, müssen entsprechend modifiziert werden. (UNUMGÄNGLICH)

Viele Programme, die für die Version 3.0 geschrieben wurden, dürften mit geringen Modifikationen auch unter der Version 6.0 compilierbar sein. Trotz-

dem ist diese Liste nicht vollständig – nicht zuletzt deshalb, weil viele Programmierer undokumentierte (oder eigentlich gar nicht beabsichtigte) Möglichkeiten der Version 3.0 ausgenutzt haben.

B. Unterschiede zu ANSI-Pascal

Die in den folgenden Abschnitten aufgeführten Unterschiede beziehen sich auf den Standard ANSI/IEEE770X3.97-1983, der in der Publikation *American Standard Pascal Computer Language* (ISBN 0-471-88944-X, herausgegeben vom Institute of Electrical and Electronics Engineers, New York) niedergelegt ist.

Abweichungen von den ANSI-Forderungen

Turbo Pascal entspricht den Anforderungen der ANSI/IEEE770X3.97-1983 mit Ausnahme der folgenden Punkte:

- ANSI-Pascal verwendet Bezeichner mit beliebiger Länge, sämtliche Zeichen sind signifikant. Turbo Pascal erlaubt ebenfalls Bezeichner beliebiger Länge, unterscheidet jedoch nur die ersten 63 Zeichen.

- ANSI-Pascal behandelt das Zeichen @ als Alternative für das Caret (^). Turbo Pascal behandelt @ als Operator mit einer eigenen Bedeutung, die nie mit der des Carets identisch ist.

- ANSI-Pascal erlaubt Kommentare, die mit { beginnen und mit *) enden oder mit (* beginnen und mit } enden. In Turbo Pascal sind diese Zeichen alternativ, aber nicht synonym: ein Kommentar muß entweder mit geschweiften Klammern oder mit (* *) eingeschlossen sein, eine Mischung beider Formen ist nicht zulässig.

- ANSI-Pascal fordert, daß jeder der möglichen Typen eines varianten Records einmal erscheint. Turbo Pascal hat diese Beschränkung nicht.

- ANSI-Pascal erlaubt keine strukturierten Komponenten von Dateitypen, die ihrerseits wieder Dateitypen als Komponenten enthalten. Turbo Pascal hat diese Beschränkung nicht.

- ANSI-Pascal ordnet einer Dateivariablen eine Puffervariable zu, deren Adressierung durch ein auf den Dateivariablennamen folgendes Caret (^) notiert wird. Dateivariablen von Turbo Pascal haben keinen Puffer zugeordnet, auf ihren Namen folgende Carets werden als Fehler ausgewiesen.

- ANSI-Pascal fordert, daß der Anweisungsteil einer Funktion mindestens eine Wertzuweisung zum Funktionsnamen enthalten muß. Turbo Pascal hat diese Beschränkung nicht.

- ANSI-Pascal erlaubt Felder, die in einem varianten Record als Selektor dienen, nicht als aktuelle Parameter. Turbo Pascal hat diese Beschränkung nicht.

- ANSI-Pascal erlaubt nicht, daß Komponenten von Variablen eines gepackten Typs als Variablenparameter übergeben werden. Turbo Pascal hat diese Beschränkung nicht.

- ANSI-Pascal deklariert prozedurale und funktionale Parameter durch Einfügen einer Prozedur- bzw. Funktionsdeklaration in die formale Parameterliste der entsprechenden Routine. Turbo Pascal implementiert prozedurale und funktionale Parameter über eigene Variablentypen, die bei ihrer Deklaration als formale Parameter dieselbe Syntax wie andere Variablentypen haben.

- ANSI-Pascal verlangt keine vorherige Initialisierung der Datei-Variablen für die Standard-Prozeduren *Reset* und *Rewrite*. In Turbo Pascal muß eine Datei-Variable vor der Verwendung beider Prozeduren mit *Assign* der entsprechenden externen Datei zugeordnet werden.

- ANSI-Pascal definiert die Standardprozeduren *Get* und *Put* zum Lesen und Schreiben von Dateien. Diese Prozeduren sind in Turbo Pascal nicht definiert.

- ANSI-Pascal definiert die Standardprozeduren *Read* und *Write* in Abhängigkeit von *Get* und *Put*, d.h. als Zugriffe auf zuvor gesetzte Puffervariablen. In Turbo Pascal haben *Read* und *Write* dieselbe Wirkung, sind aber automatische Operationen.

- ANSI-Pascal erlaubt die Verwendung der Standardprozedur *New* mit der Syntax *New*(*p*,*c1*,..,*cn*) zur Erzeugung einer dynamischen Variablen mit einem varianten Teil bestimmten Typs. Turbo Pascal kennt diese Verwendungsmöglichkeit von *New* nicht.

- ANSI-Pascal erlaubt die Verwendung der Standardprozedur *Dispose* mit der Syntax *Dispose*(*p*,*c1*,..,*cn*) zur Entfernung einer dynamischen Variablen mit einem varianten Teil bestimmten Typs. Turbo Pascal kennt diese Verwendungsmöglichkeit von *Dispose* nicht.

- ANSI-Pascal definiert die Standardprozeduren *Pack* und *Unpack* zum »Packen« und »Entpacken« von Variablen. Turbo Pascal enthält diese beiden Prozeduren nicht.

- ANSI-Pascal liefert für den Term *i* **mod** *j* immer ein positives Ergebnis zurück bzw. weist negative Ergebnisse als Fehler aus. Turbo Pascal behandelt diesen Term als *i* - (*i* **div** *j*) * *j* und liefert auch negative Werte ohne Fehlermeldung zurück.

- ANSI-Pascal erlaubt **goto**-Anweisungen, deren Sprungziel in einem Block liegt, der den Block des **goto** einschließt, d.h. ihm übergeordnet ist. Turbo Pascal weist alle Sprünge, die aus einem Block hinausführen, als Fehler aus.

- ANSI-Pascal erzeugt einen Laufzeitfehler, wenn sämtliche Vergleiche eines **case**-Blocks fehlschlagen. Turbo Pascal ignoriert den **case**-Block in einem solchen Fall vollständig, solange er nicht einen mit **else** eingeleiteten Zweig enthält; ein Laufzeitfehler wird in keinem Fall erzeugt.

- ANSI-Pascal weist Möglichkeiten, die Laufvariable einer **for**-Schleife zu verändern (z.B. ihre Übergabe als **var**-Parameter) als Fehler aus. Turbo Pascal hat diese Beschränkung nicht.

- ANSI-Pascal ordnet Variablen vom Typ *Char* bei einem *Read* aus einer Textdatei ein Leerzeichen zu, wenn das Zeilenende bereits vor der *Read*-Anweisung erreicht war (*Eoln = True*). Turbo Pascal ordnet der Variablen in diesem Fall ein Zeilenende (ASCII-Code 13) zu.

- ANSI-Pascal beendet den Lesevorgang einer numerischen Variablen aus einer Textdatei, sobald ein nicht-numerisches Zeichen gelesen wird. Turbo Pascal beendet derartige Lesevorgänge nur bei Leerzeichen oder Steuerzeichen (inklusive einem Zeilenvorschub).

- ANSI-Pascal schneidet gepackte Stringvariablen bei einem *Write* in eine Textdatei automatisch ab, wenn die Stringlänge größer als die gesetzte Feld/Recordlänge ist. Turbo Pascal schreibt Strings immer in voller Länge, selbst wenn dabei Feld- bzw. Recordgrenzen überschritten werden.

- ANSI-Pascal definiert die Standardprozedur Page, die bei einer Ausgabe in eine Textdatei einen Seitenumbruch bewirkt. In Turbo Pascal ist diese Prozedur nicht definiert. Äquivalent zu Page(F) kann Write(F,Chr(12)) verwendet werden.

Hinweis: Außer im Falle nicht definierter Routinen kann Turbo Pascal nicht bestimmen, ob ein Programm eine oder mehrere der hier aufgeführten Abweichungen unberücksichtigt läßt.

Erweiterungen des ANSI-Standards

Die folgenden Punkte stellen Erweiterungen des durch ANSI/IEEE770X3.97-1983 gesetzten Standards dar:

- Turbo Pascal reserviert über den Standard hinaus die folgenden Bezeichner:

absolute	**inline**	**shr**
assembler	**interface**	**string**
constuctor	**interrupt**	**unit**
destruktor	**object**	**virtual**
external	**private**	**uses**
far	**near**	**xor**
implementation	**shl**	

- Ein Bezeichner kann zusätzlich Unterstriche (_) enthalten.

- Integer-Konstanten können durch Voranstellen eines Dollarzeichens ($) in hexadezimaler Notation angegeben werden.
- Die Verwendung von Bezeichnern als Labels ist zulässig.
- String-Konstanten sind mit den String-Typen von Turbo Pascal kompatibel; sie können Steuer- und Grafikzeichen enthalten.
- Deklarationen von Funktionen, Prozeduren, Variablen, Typen, Konstanten und Labels sind in beliebiger Folge und wiederholt innerhalb eines Blocks möglich.
- Bezeichner für (vorzeichenbehaftete) Konstanten können sowohl für Werte im Integerformat als auch für *LongInt* oder *Extended* stehen.
- Turbo Pascal implementiert die zusätzlichen Integertypen *ShortInt*, *Byte*, *Word* und *LongInt* sowie die zusätzlichen Real-Typen *Single*, *Double*, *Comp* und *Extended*.
- Arithmetische Operationen mit Integern erzeugen Resultate vom Typ Integer, gemischte Operationen mit Integern und *LongInts* haben den Ergebnistyp *LongInt*. *LongInt*-Variablen sind mit Integervariablen kompatibel, solange ihr Wert innerhalb der Integergrenzen liegt.
- Arithmetische Operationen mit Real-Typen oder gemischte Operationen mit Real- und Integertypen haben den Ergebnistyp *Extended*, der mit den Typen *Real*, *Double* und *Comp* kompatibel ist, solange der Wert innerhalb der entsprechenden Grenzen liegt.
- Turbo Pascal implementiert String-Variablen, die sich von gepackten ANSI-Strings durch ein vorangestelltes Längenbyte und die Möglichkeit der dynamischen Längenänderung während der Laufzeit des Programms unterscheiden.
- Turbo Pascal implementiert prozedurale und funktionale Typen, die sich einerseits als funktionale bzw. prozedurale Parameter verwenden lassen; andererseits können über sie Variablen entsprechenden Typs deklariert werden.
- Turbo Pascal implementiert typisierte Konstanten, die zu Beginn des Programms initialisiert werden und sich ansonsten wie Variablen des entsprechenden Typs verhalten. Variablen beliebigen Typs (außer Datei-Variablen) lassen sich auf diese Weise direkt initialisieren.
- Die Regeln zur Typ-Kompatibilität wurden dahingehend erweitert, daß *Char*- und gepackte String-Typen mit normalen String-Typen verträglich sind.
- Über die Behandlung von String-Variablen als Array ist der Zugriff auf einzelne Zeichen der Variablen möglich.
- Über den Zusatz **absolute** können Variablen auf einen bestimmten Speicherbereich festgelegt werden.
- Ein Bezug auf eine Variable kann eine Funktion enthalten, deren Ergebnis ein Zeiger-Typ ist. Das Ergebnis dieser Funktion wird automatisch in die tatsächliche Speicheradresse der entsprechenden dynamischen Variablen umgewandelt (»de-referencing«).

- Der Typ einer Variablen kann für die Dauer eines Zugriffs über eine Typ-Umwandlung verändert werden.
- Turbo Pascal implementiert typisierte Konstanten, die zur Deklaration initialisierter Variablen aller Typen, aber nicht von Dateitypen, verwendet werden können.
- Turbo Pascal implementiert drei logische Operatoren, die im ANSI-Standard nicht enthalten sind: **xor**, **shl** und **shr**.
- Die Operatoren **not**, **and**, **or** und **xor** können zusammen mit Integer-Operanden verwendet werden und erzeugen in diesem Fall eine bitweise logische Verknüpfung.
- Der Operator + verbindet zwei Stringteile miteinander.
- Turbo Pascal erlaubt den Vergleich von String-Variablen über relationale Operatoren.
- Turbo Pascal implementiert den Operator @, über den die Speicheradresse einer Variablen, einer Prozedur oder einer Funktion ermittelt werden kann.
- Der Ergebnistyp eines Ausdrucks kann durch eine explizite Typ-Umwandlung verändert werden.
- Innerhalb eines **case**-Blocks sind Wertebereiche erlaubt, zusätzlich ist ein **else**-Zweig definierbar.
- Prozeduren und Funktionen können als **external** (Unterprogramme in Maschinensprache), als **inline** (im Quelltext definierter Maschinencode) oder als **interrupt** (Interrupt-Behandlung) deklariert werden.
- Ein **var**-Parameter kann untypisiert sein. In diesem Fall kann jede Variablen-Adressierung als aktueller Parameter verwendet werden.
- Turbo Pascal implementiert separat compilierbare Units zur Vereinfachung der Modularisierung eines Programms.
- Turbo Pascal implementiert die folgenden Prozeduren und Funktionen zur Dateibearbeitung, die nicht in ANSI-Pascal enthalten sind:

Append	FilePos	Rename
BlockRead	FileSize	RmDir
BlockWrite	Flush	Seek
ChDir	GetDir	SeekEof
Close	Mkdir	SeekEoln
Erase		

- String-Variablen können mit den Standardprozeduren *Read*, *Readln*, *Write* und *Writeln* gelesen bzw. geschrieben werden.
- Turbo Pascal implementiert die folgenden Standardprozeduren und -funktionen als Erweiterung des ANSI-Standards:

Addr	GetMem	Move	RunError
Concat	Halt	Ofs	Seg

Copy	Hi	ParamCount	SPtr
CSeg	Inc	ParamStr	SizeOf
DSeg	Insert	Pi	SSeg
Dec	Int	Pos	Str
Delete	Length	Ptr	Swap
Exit	Lo	Random	TypeOf
FillChar	Mark	Randomize	UpCase
Frac	MaxAvail	Release	Val
FreeMem	MemAvail		

- Turbo Pascal implementiert weitere Standardprozeduren, -funktionen, Datentypen und Konstanten über Standard-Units.

Hinweis: Turbo Pascal verfügt über keine Mechanismen, mit denen eine Benutzung dieser Erweiterungen erkannt werden könnte.

Abhängigkeiten von der Implementation

Wie in ANSI/IEEE770X3.97-1983 festgelegt, ist die Wirkung implementationsabhängiger Befehlsfolgen in Pascal undefiniert. Programme sollten grundsätzlich so geschrieben sein, daß die Ergebnisse von Befehlen nicht auf der Reihenfolge der Umsetzung durch den Compiler beruhen. Bei den folgenden Operationen ist die zeitliche Abfolge der Rechenschritte von der Implementation des Compilers abhängig:

- Berechnung von Index-Ausdrücken bei einem Variablenzugriff;
- Berechnung der Ausdrücke in Mengenoperationen (Sets);
- Reihenfolge der Auswertung bei binären Operatoren;
- Auswertung der aktuellen Parameter bei einem Funktionsaufruf;
- Auswertung der linken und rechten Seite einer Zuweisung;
- Auswertung der aktuellen Parameter bei einem Prozeduraufruf.

Zusätzlich von der Implementation des Compilers abhängig sind:

- das Lesen einer Textdatei, die unter Verwendung der Prozedur *Page* beschrieben wurde;
- die Anbindung von Konstrukten außerhalb des Programms durch Variablen, die innerhalb des Programms definiert wurden.

Behandlung von Fehlern

Die folgende Aufstellung gibt Fehlermöglichkeiten wieder, die in Anhang D des ANSI-Standards definiert sind, von Turbo Pascal aber nicht automatisch erkannt werden. Die ANSI-Fehler 6, 19 bis 22 und 25 bis 31 werden von Turbo Pascal ebenfalls nicht erkannt, können aber aufgrund des unterschiedlichen Sprachumfangs hier auch nicht auftreten.

2 Wenn *t* den Selektor eines varianten Record-Feldes darstellt und *f* eine aktive Variante dieses Feldes ist, dann kann *t* nicht geändert werden, solange eine Referenz zu *f* existiert. In Turbo Pascal wird dieser Fehler nicht erkannt.

3 Wenn *p* eine Zeigervariable ist und den Wert **nil** enthält, sollte ein Zugriff über *p*^ als Fehler ausgewiesen werden. In Turbo Pascal bleibt dieser Fehler unerkannt.

4 Wenn *p* eine Zeigervariable und undefiniert ist, sollte der Prozeduraufruf *Dispose*(*p*) als Fehler ausgewiesen werden. Dieser Fehler wird in Turbo Pascal nicht erkannt.

5 Wenn *p* eine Zeigervariable ist und eine Referenz via *p*^ existiert, sollte eine Veränderung von *p* als Fehler ausgewiesen werden. In Turbo Pascal wird dieser Fehler nicht erkannt.

42 Der Funktionsaufruf *Eoln*(*f*) erzeugt in ANSI-Pascal einen Fehler, wenn *Eof*(*f*) *True* ist. In Turbo Pascal liefert *Eoln*(*f*) den Wert *True* zurück, wenn das Dateiende erreicht ist.

43 Das Auslesen einer nicht definierten Variablen, d.h. ihre Verwendung innerhalb eines Ausdrucks, sollte einen Fehler erzeugen. Dieser Fehler bleibt in Turbo Pascal unerkannt.

46 ANSI-Pascal erzeugt einen Fehler, wenn das Ergebnis des Terms *i* **mod** *j* kleiner oder gleich Null ist. Turbo Pascal liefert auch Null oder negative Werte zurück, ohne dabei einen Fehler zu erzeugen.

48 Im Anweisungsteil einer Funktion sollte mindestens eine Zuweisung zum Funktionsnamen, d.h. zum Ergebnis der Funktion, stattfinden, ansonsten weist ANSI-Pascal einen Fehler aus. Turbo Pascal bemerkt diesen Fehler nicht und liefert einen undefinierten Funktionswert zurück.

51 Wenn sämtliche Prüfungen eines case-Blocks fehlschlagen, erzeugt ANSI-Pascal einen Fehler. Turbo Pascal ignoriert in diesem Fall den case-Block vollständig bzw. führt den mit else definierten Zweig aus, der in ANSI-Pascal nicht definiert ist.

Index

A
Abbruch
  von MAKE.EXE 249
  Programmen 136
Abhängigkeiten
  von der Implementation 298
absolute Variablen
  Unterschiede zur Version 3.0 277
Addr (Funktion)
  Ersatz durch @ 276
Adress-Operator 41
Aktionsschalter
  Cancel 15
  Help 15
  OK 15
Allgemeine Tastenkürzel 9
ANSI-Pascal, Unterschiede 293
Anwahl der Menü-Befehle 8
Anweisungen
  case 44
  for 48
  if 43
  repeat 46
  Schleifen 45
  while 45
  Wiederholung 45
Anweisungsblöcke 43
Arrays 97
  in Watch-Ausdrücken 152
  Index-Prüfung 130
Aufruf 168 f.
  von BINOBJ 223
  von GREP 225
  von MAKE 248
  von THELP 253
  von TOUCH 258
Aufrufe
  Verfolgen (Debugger) 138
  von Prozeduren und Funktionen 52
  von Unterprogrammen 52
  Unterschiede zur Version 3.0 283
Aufzählungstypen 99
  in Watch-Ausdrücken 152
Ausdrücke
  für GREP 227
  Operatoren 37
  Watch 141, 151
  Unterschiede zur Version 3.0 282
Ausführung, bedingte 42
Ausgabe 28
  Begriffsdefinition 28
  Fehlerprüfung 55
  Feldbreite 29
  Formate von Evaluate und Watch 153
  formatierte 28
  Konvertierung 288
  von Textdateien 106
  Writeln-Prozedur 28
Ausgabe-Fenster 197

B
BCD-Arithmetik
  Unterschiede zur Version 3.0 277
bedingte
  Ausführung 42
  Compilierung 122
Bedingungen für Breakpoints 187
Begrenzerpaare 209
Beispiele
  für explizite Formatangaben 155
  für Grafik 24
  für GREP 229
  für Overlays 88
  für Unit Crt 110
Bereichs-Überprüfung
  Beispielprogramm 130
Bewährung 78

Bezeichner
  Benennung 36
  lokale 53
  mehrfach deklarierte 68
  qualifizierte 68, 157
  Konvertierung 285
  Unterschiede zur Version 3.0 275
Bibliotheks-Units als Overlays 79
Bildschirm
  Prozeduren und Funktionen 110
binäre Operatoren 38
BINOBJ.EXE 223
Bitmanipulation 38
Blöcke
  von Funktionen und Prozeduren 50
blockstrukturierte Anweisungen 43
Boolean (Datentyp) 35
  in Watch-Ausdrücken 152
  Unterschiede zur Version 3.0 282
Breakpoints 186
  verlieren 187
Build 183, 270
  Arbeitsweise 122
BUILTINS.MAK 249

C
case-Anweisung 44
  und Strings 277
CGA-Karte 169
Char (Datentyp) 33
  in Watch-Ausdrücken 152
  Steuerzeichen 33
Close (Prozedur)
  Unterschiede zur Version 3.0 277
Compile (Menü Compile) 270
Compile-Menü 182
Compile/Destination 183
Compiler
  bedingte Compilierung 122
  Funktion 19
  in der IDE 7
  Kommandozeilen-Version 7, 211
  separate Kompilierung 269
Compiler-Befehle
  Bereichsüberprüfung 130
  Ein-/Ausgabeprüfung 55
  für Overlays 78
  für TPC 212, 214
  Konvertierung 290
  Vergleich mit Version 3.0 272
Compiler-Optionen 188
Crt (Standard-Unit)
  Beispielprogramm 110
  nicht als Overlay 93
CSeg (Funktion)
  Unterschiede zur Version 3.0 277
Ctrl-Break 136

D
Datei öffnen 171
Datei-Typen
  Unterschiede 105
Dateien 104
  Begriffsdefinition 104
  Standard- 105
  Text- 105
  typisierte 106
  untypisierte 107
Dateien speichern 172
Dateinamen
  für GREP 226
Dateivariablen
  in Watch-Ausdrücken 153
Datentypen 30
  Boolean 35
  grundlegende 30
  Integer 31
  Konvertierung 286
  Real- 31
  selbstdefinierte 98
  String 33
  Zeiger 35
DateTime (Record-Typ)
  Beispiel 109
Debug-Informationen 191
Debug-Menü 184
Debugger 132
  Abbruch mit Ctrl-Break 136
  Programmier-Regeln 158
  Tastenkürzel 11
  Tutorial 133
  und Overlays 93
Deklaration
  des Programmnamens 271
  mehrfache 68
Dialogfenster 15
  Aktionsschalter 15
  Eingabefelder 15
  Listen 15
  Markierungsfelder 15
  Schaltfelder 15
Direktiven in MAKE-Dateien 244
Display swapping 191

DOS
  Befehle 174
  Funktionsaufrufe 108
Dos (Bibliotheks-Unit)
  als Overlay 79
  Anwendung 108
Double(Datentyp)
  in Watch-Ausdrücken 152
Dual-Monitor-Modus 168
dynamische Speicherverwaltung
  Beispielprogramm 100

E
Eckfeld zur Änderung der
  Fenstergröße 12
Edit-Menü 175
Edit-Tastenkürzel 10
Edit/Clear 176
Edit/Copy 176
Edit/Copy Example 176
Edit/Cut 176
Edit/Paste 176
Edit/Restore line 175
Edit/Show Clipboard 176
Editor 201
  Befehle 202
  Begrenzerpaaresuchen 208
  Block-Kommandos 203 f
  Cursorbewegung 202
  Einfüge- und Lösch-Kommandos 203
  Suchen und Ersetzen 207
Editor
  Einstellungen 193
  Heap 169
EGA-Palette 170
Eingabe 41
  Fehlerprüfung 55
  Konvertierung 288
  Readln-Prozedur 41
  Unterschiede zur Version 3.0 278
Eingabeaufzeichnungsliste 16
Einstellung der Maus 9
Elemente der Programmierung 27
EMS-Karte
  für Overlays 83
Entwicklungsumgebung 7
Environment/Colors 195
Environment/Editor 193
Environment/Mouse 194
Environment/Startup 195
error-Direktive in MAKE-Dateien 247
Ersetzen 178
Erweiterungen und Verbesserungen
280
Evaluate (Menü Debug)
  Anwendungsbeispiel 140
  explizite Formatangaben 153
  Syntax 151
Evaluate/modify 184
.EXE-Datei 183
Exit-Prozeduren
  und der Debugger 160
Expansionsspeicher 170
explizite Abhängigkeiten 233
Extended (Datentyp)
  in Watch-Ausdrücken 152

F
{$F+} (Compiler-Befehl) 87
Fehler
  Arten 127
  Korrektur und Neustart 145
  Laufzeit 132
  logische 128
  Meldungen 247, 250
  Prüfverfahren 163
  Syntax- 127
  Unterschiede zur Version 3.0 282
Fehlerbehandlung
  ANSI-Standard 299
  IOResult-Funktion (Beispiel) 56
  Konvertierung 290
Fehlersuche 127
  Laufzeitfehler 128
  und Overlays 93
  Unterschiede zur Version 3.0 283
Feldbreite (Write) 29
Fenster 11
  Befehle 196
  Puffer 170
  schließen 12
  Tastenkürzel 10,12
  vergrößern und verkleinern 12
Fensterelemente 11 f.
Fensternummer 12 f.
File (Datentyp)
  in Watch-Ausdrücken 153
File-Menü 171
File/DOS Shell 174
File/Exit 175
File/New 172
File/Open 172
File/Print 174
File/Save 173

File/Save all 173
File/Save as 173
FindFirst (Prozedur)
  Beispiel 109
FindNext (Prozedur)
  Beispiel 109
for-Schleifen 48
  Unterschiede zur Version 3.0 278
Formatangaben in
  Debugger-Ausdrücken 184
Formate
  explizite Angaben (Beispiele) 155
  für Evaluate und Watch 153
formatierte Ausgabe 28
Funktionen 49
  Begriffsdefinition 50
  Ergebnis 51
  Deklaration, suche nach 147
  Unterschiede zur Version 3.0 283

G
Go to cursor (Menü Run)
  Anwendungsbeispiel 139
Grafik
  Beispielprogramm 24
Grafik-Bildschirm 169
Grafiktreiber
  als .OBJ-Datei 223
Grafikzeichen als Char-Konstanten 33
Graph (Unit) 93
Graph3 (Standard-Unit) 283
GREP 225
  Aufruf-Beispiele 229
  Schalter 226
  Suchstrings 227
große Programme und Units 73
grundlegende Datentypen 30

H
Heap-Bereich und Overlays 79
Help-Menü 198
Help/Contents 199
Help/Index 199
Help/Topic search 199
Hilfe über Hilfe 7
Hilfe-System 7
  Tastenkürzel 11
Hilfebildschirm 198
Hilfestellung 7
Hinzufügen von
  Watch-Ausdrücken 142

I
IDE 7
IDE-Einstellungen 193
if-Anweisung 43
Implementation
  Abhängigkeiten 298
Implementations-Teil eines Units 62
implizite Abhängigkeiten 236
Indirektions-Operator 41
Initialisierung
  der Overlay-Verwaltung 83
  von Units 63
Initialisierungs-Teil
  in Overlays 92
  Reihenfolge 64
inline
  Konvertierung 29
  und der Debugger 160
  Unterschiede zur Version 3.0 282
Input (Standard-Datei) 104
Installation 2
Integer (Datentyp) 31
  in Watch-Ausdrücken 152
  Unterschiede zur Version 3.0 279
Integrierte Entwicklungsumgebung 7
  Unterschiede zur Kommandozeilen-Version 269
Interface-Teil einer Unit 61
Interrupts
  Unterschiede zur Version 3.0 282
IOResult (Funktion)
  Beispiel 56

K
Kommandozeilen-Optionen 168
Kommandozeilen-Parameter
  Quelltext-Dateiname 18
  von MAKE 248
Kommandozeilen-Version des
  Compilers 211
  Unterschiede zur integrierten Entwicklungsumgebung 269
Kommentare 54
  in MAKE-Dateien 233
  verschachtelte 55
Konfigurationsdatei 168
Konstanten
  Unterschiede zur Version 3.0 279
Konvertierung
  Compiler-Befehle 290
  Datentypen 286
  Ein- und Ausgabe 288

Fehlerprüfung 290
Maschinensprache-Routinen 291
Programm-Organisation 289
Speicherbelegung 289
Sprachstruktur 288
Units Turbo3, Graph3 283
von 3.0-Programmen 268
vordefinierte Bezeichner 285

L
Labels
Unterschiede zur Version 3.0 279
LastMode (Variable) 276
Laufvariable 48
Laufzeitfehler 128, 132, 179
Unterschiede zur Version 3.0 282
Linker
Compiler-Schalter 220
Listen, verkettete 101
logische Fehler 128
logische Operatoren 40
lokale Bezeichner 53
lokale Variablen 52
Löschen von Watch-Ausdrücken 142

M
Make (Menü Compile) 270
Arbeitsweise 121
MAKE.EXE 232
'Konfigurationsdatei'
BUILTINS.MAK 249
Abbruchbedingungen 249
Aufruf 248
Beispiel 123
Direktiven 244
explizite Regeln 233
Fehlermeldungen 250
Hilfsprogramm TOUCH 258
implizite Regeln 236
Kommandolisten 238
Makros 239, 241
Operatoren 246
Makros in MAKE-Dateien 239
Map file (Compiler-Schalter) 220
Markieren von Text 175
Markierungsfelder
in Dialogfenstern 15
Maschinensprache
Konvertierung 291
und Overlays 94
Unterschiede zur Version 3.0 282
Maus-Einstellungen 9, 194
Mem, MemW, MemL (Pseudo-Arrays)
Unterschiede zur Version 3.0 282
Mengen
in Watch-Ausdrücken 153
Operatoren 41
Menü-Befehle 8
Anwahl der 8
mit Fortsetzungspunkten 15
Menü-Tastenkürzel 10
Menüzeile 8
modulare Compilierung 269

N
New (Prozedur)
Beispiel 102

O
{$O+} (Compiler-Befehl) 86
Online-Hilfesystem 167
Operationen eines Programms 27
Operatoren 37
Adress- 41
binäre 38
Bitmanipulation 38
in MAKE-Dateien 246
logische 40
Mengen- 41
Rangfolge 37
relationale 39
String- 41
unäre 38
Zuweisung 37
Options
Linker 190
Memory Sizes 190
Options-Menü 188
Options/Compiler 188
Options/Debugger 191
Options/Directories 192
Options/Environment 193
Organisation von Programmen 117
Output (Standard-Datei) 104
Overlay (Unit)
nicht als Overlay 93
Overlay-Prozeduren
OvrGetRetry 85
OvrSetRetry 85
Overlay-Puffer 76, 169
Größe ermitteln 84
Größe festlegen 84
Laden und Freigeben 76
Minimalgröße 75

und Programmgröße 79
Zurücksetzen 84
Overlay-Variablen
OvrFileMode 81
OvrLoadCount 81
OvrReadBuf 81
OvrTrapCount 81
Overlays 74
Aufrufe über Zeiger 93
Beispiele 88
Bewährungsphase 78
Compiler-Befehle 78
EMS 82
in EXE-Dateien 85
praktische Voraussetzungen 78
Programmierung mit 86
Puffer 76
Puffer laden und freigeben 76
Puffer-Optimierung 77
technische Hintergründe 75
und Assembler-Routinen 94
und der Debugger 93
wann nicht 93
OvrClearBuf (Prozedur) 84
OvrFileMode 81
OvrGetBuf (Funktion) 84
OvrGetRetry 85
OvrInit (Prozedur) 83
OvrInitEMS (Prozedur) 83
OvrLoadCount 81
OvrReadBuf 81
OvrSetBuf (Prozedur) 84
OvrSetRetry 85
OvrTrapCount 81

P
Parameter
für TPC 212
prozedurale 93
Platzprobleme mit dem Debugger 161
Pointer (Datentyp) 35
in Watch-Ausdrücken 152
Position und Größe 196
Positionsmarke 13
praktische Voraussetzungen
für Overlays 78
Program-Deklaration 271
Programm
Abbruch mit Ctrl-Break 136
Abhängigkeiten 121
Arbeit mit Overlays 86
grundlegende Elemente 27
Kommentierung 54
Konvertierung von 3.0 268, 289
Neustart nach Korrekturen 145
mit MAKE.EXE 123
Organisation 117
prüfen 163
Speicherung 20
starten 180
Struktur 50
Programmiertechniken
Unterschiede zur Version 3.0 276
Prozedur-Variablen
Anwendungsbeispiel 113
Prozeduren 49
Ausgabe- 28
Begriffsdefinition 50
Eingabe- 41
Suche nach 147
Unterschiede zur Version 3.0 283
Puffer
Overlay 76
Puffergröße von Textdateien
Unterschiede zur Version 3.0 280

Q
qualifizierte Bezeichner 68, 157
Quelltext
als Kommandozeilen-Parameter 18
Vergleich mit GREP 225

R
Read (Prozedur)
in typisierten Dateien 107
ReadKey (Funktion)
Beispiel 111
Readln (Prozedur) 41
Unterschiede zur Version 3.0 278
Real (Datentyp)
in Watch-Ausdrücken 152
Real-Typen 31
Unterschiede zur Version 3.0 281
Records 99
in typisierten Dateien 106
in verketteten Listen 102
in Watch-Ausdrücken 152
Register-Fenster 198
Reihenfolge von
Initialisierungs-Teilen 64
relationale Operatoren 39
repeat-Anweisung 46
Rollbalken 12
Positionsmarke 13

Rumpf einer Schleife 46
Run-Tastenkürzel 11
Run-Menü 180
Run/Parameters 180, 182
Run/Program reset 181
Run/Run 180
Run/Trace into 181
Runtime Errors 189

S
Schalter
  für GREP 226
  für TPC 212
Schaltfelder
  Eingabeaufzeichnungsliste 16
  in Dialogfenstern 15
Schleifen 45
  for 48
  repeat..until 46
  Unterschiede zur Version 3.0 278
  while 45
schrittweise verfolgen 181
Search-Menü 176
Search/Find error 179
SearchRec (Record-Typ)
  Beispiel 109
selbstgeschriebene Units 66
Selektor 44
separate Compilierung 269
Single (Datentyp)
  in Watch-Ausdrücken 152
Speicher
  Platzprobleme 161
Speicherbelegung
  Unterschiede zur Version 3.0 289
Speichern unter 173
Speicherung
  von Programmen 20
Standard-Dateien 105
Standard-Units
  als Overlays 79
  Begriffsdefinition 59
Start von
  BINOBJ 223
  GREP 225 f.
  MAKE 248
  THELP 253
  TOUCH 258
  TURBO 17
Startbalken 136
Startposition 136
Statuszeile 14
Steuerzeichen als Char-Konstanten 33
String (Datentyp) 33
String-Typen
  Operatoren 41
  Unterschiede zur Version 3.0 281
Strings
  Begriffsdefinition 33
  in Watch-Ausdrücken 152
  Unterschiede zur Version 3.0 277
Struktur
  eines Programms 50
  von Units 60
Suchen 178
  nach Deklarationen 147
Suchen wiederholen 178
Suchen-Optionen 177
Swap-Datei 170
Syntax options 189
Syntax-Hilfe
  für TPC 211
Syntax-Prüfung 189
System (Standard-Unit)
  Initialisierung 104
System (Unit)
  nicht als Overlay 93
System-Menü 171

T
Tastatur
  direktes Lesen 111
  Unterschiede zur Version 3.0 279
Tastenkürzel 9
  Allgemeine 9
  des Debuggers 149
  Edit- 10
  für das integrierte Hilfe-System 11
  für Debugger und Run 11
  Menü- 10
  zur Fensterverwaltung 10
Text suchen 176
Textdateien 105
  Unterschiede zur Version 3.0 280
Texteingabe 201
THELP (Zusatzprogramm) 253
  Aktivierung 254
  Entfernen 255
  Kommandozeilen-Parameter 255
  Start 253
  Tastenbefehle 254
Titel 12
Toggle Breakpoint 186
TOUCH (Hilfsprogramm) 258

TPC.EXE
  und Compiler-Befehle 214
  und der Turbo Debugger 219
  Unterschiede zu TURBO.EXE 269
.TPL-Dateien
  TURBO.TPL 259
TPUMOVER 259
  Kommandozeilen-Parameter 260
Trace into (Menü Run)
  Anwendungsbeispiel 138
Turbo Debugger
  TPC-Parameter 219
Turbo Pascal
  Debugger 132
  Erweiterungen des Sprachstandards 295
  Implementation 298
  Kommandozeilen-Version 211
  Start mit Dateinamen 18
  Version 3.0 (Unterschiede) 261
Turbo Pascal Version 3.0
  Compiler-Befehle 272
  Konvertierung 268
TURBO.EXE
  Unterschiede zu TPC.EXE 269
TURBO.TPL (Unit-Bibliothek) 170, 259
Turbo3 (Standard-Unit) 283
  nicht als Overlay 93
Typ-Umwandlung
  Unterschiede zur Version 3.0 280
Typen
  selbstdefinierte 98
  Unterschiede zur Version 3.0 278
typisierte Dateien 106

U
unäre Operatoren 38
Units
  Abhängigkeiten 70
  gemeinsame Datendeklarationen 72
  Implementations-Teil 70
  mehrfache Aufnahme 65
  Standard 65
Units (s.a. Standard-Units) 269
  Abhängigkeiten 121
  als Overlays 79
  Begriffsdefinition 59
  Dateitypen .TPU und .TPL 259
  gemeinsame Variablen 117
  Implementations-Teil 62
  Initialisierung 63
  Interface-Teil 61
  mehrfach deklarierte Bezeichner 68
  Programmorganisation 117
  Prüfsumme 121
  Recompilierung 121
  selbstgeschriebene 66
  Struktur 60
  und große Programme 73
UnpackTime (Prozedur)
  Beispiel 109
Unterprogramme (s.a. Funktionen, Prozeduren) 49
Unterschiede zu ANSI-Pascal 293
Unterschiede zur Version 3.0 261
  BCD-Arithmetik 277
  Char-Typen und Strings 277
  Close-Prozedur 277
  CSeg und DSeg 277
  direkte Kontrolle der Tastatur 279
  Eingaben von Integern 278
  Erweiterungen und Verbesserungen 280
  for-Laufvariablen 278
  Integerwerte 279
  Labels 279
  Programmiertechniken 276
  typisierte Konstanten 279
untypisierte Dateien 107
uses-Anweisung
  mehrfache für ein Unit 65

V
Variablen
  Ausgabe 140
  dynamische 100
  globale 100
  lokale 52
  verändern 156
  Unterschiede zur Version 3.0 275, 284
Verändern von Variablenwerten 156
Vergleich von Quelltexten 225
Verlassen von Turbo Pascal 171
verschachtelte Kommentare 55
Verwaltung von Projekten 117
Verzeichnis
  absuchen 109
  wechseln 173
Verzeichnisnamen 192

W
Watch-Ausdrücke
  Anwendungsbeispiel 141
  bearbeiten und löschen 142

Begriffsdefinition 141
explizite Formatangaben 153
hinzufügen 142
Syntax 151
Watch-Fenster 197
Watches 185
while-Anweisung 45
Window-Menü 196
Window/Call Stack 197
Window/Cascade 197
Window/Close 197
Window/Output 197
Window/Registers 198
Window/Tile 196
Window/User screen 197
Window/Zoom 196
Write (Prozedur)
in typisierten Dateien 107
Writeln (Prozedur) 28

Z
Zeichensätze als .OBJ-Datei 223
Zeiger
Aufrufe über 93
Beispielprogramm 100
in Watch-Ausdrücken 152
und Overlays 93
Zeiger-Typen 35
Zoom-Feld 12
Zugriff
auf mehrfach deklarierte Bezeichner 69
Zusatzprogramme 223
BINOBJ 223
GREP 225
THELP 253
TOUCH 258
Zuweisungen
Operatoren 37
Zwischenablage-Fenster 176

Gestärkt durch die Leistungsfähigkeit und Effektivität der objektorientierten Programmierung gleibt die Stellung von Turbo Pascal als Marktführer auch in Zukunft gesichert... Sicher ist nur eins, OOP wird neue Bereiche eröffnen, sei es in Datenbank-Management-Systemen oder auch in Programmiersprachen der Zukunft.

Jürgen Mertes/Klaus Schäfer
PC Magazin, 18. Oktober 1989

Das objektorientierte Modell hat das Zeug dazu, nach der Einführung der strukturierten Programmierung vor gut zwei Jahrzehnten die zweite »sanfte« Wende auf dem Gebiet der Software-Entwicklung zu sein.

Ronald Fischer
Toolbox, Oktober 1989

B O R L A N D